W3School 数据库教程合集

wizardforcel

Published with GitBook



目錄

W3School 数据库教程合集

来源：菜鸟教程

整理：飞龙

感谢菜鸟教程站长的翻译和奉献。

W3School SQL教程

来源：SQL教程

整理：飞龙

SQL基础

SQL 简介

SQL 是用于访问和处理数据库的标准的计算机语言。

什么是 SQL？

- SQL 指结构化查询语言
- SQL 使我们有能力访问数据库
- SQL 是一种 ANSI 的标准计算机语言

编者注：ANSI，美国国家标准化组织

SQL 能做什么？

- SQL 面向数据库执行查询
- SQL 可从数据库取回数据
- SQL 可在数据库中插入新的记录
- SQL 可更新数据库中的数据
- SQL 可从数据库删除记录
- SQL 可创建新数据库
- SQL 可在数据库中创建新表
- SQL 可在数据库中创建存储过程
- SQL 可在数据库中创建视图
- SQL 可以设置表、存储过程和视图的权限

SQL 是一种标准 - 但是...

SQL 是一门 ANSI 的标准计算机语言，用来访问和操作数据库系统。SQL 语句用于取回和更新数据库中的数据。SQL 可与数据库程序协同工作，比如 MS Access、DB2、Informix、MS SQL Server、Oracle、Sybase 以及其他数据库系统。

不幸地是，存在着很多不同版本的 SQL 语言，但是为了与 ANSI 标准相兼容，它们必须以相似的方式共同地来支持一些主要的关键词（比如 SELECT、UPDATE、DELETE、INSERT、WHERE 等等）。

注释：除了 SQL 标准之外，大部分 SQL 数据库程序都拥有它们自己的私有扩展！

在您的网站中使用 SQL

要创建发布数据库中数据的网站，您需要以下要素：

- RDBMS 数据库程序（比如 MS Access, SQL Server, MySQL）

- 服务器端脚本语言（比如 PHP 或 ASP）
- SQL
- HTML / CSS

RDBMS

RDBMS 指的是关系型数据库管理系统。

RDBMS 是 SQL 的基础，同样也是所有现代数据库系统的基础，比如 MS SQL Server, IBM DB2, Oracle, MySQL 以及 Microsoft Access。

RDBMS 中的数据存储在被称为表（tables）的数据库对象中。

表是相关的数据项的集合，它由列和行组成。

SQL 语法

数据库表

一个数据库通常包含一个或多个表。每个表由一个名字标识（例如"客户"或者"订单"）。表包含带有数据的记录（行）。

下面的例子是一个名为 "Persons" 的表：

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

上面的表包含三条记录（每一条对应一个人）和五个列（Id、姓、名、地址和城市）。

SQL 语句

您需要在数据库上执行的大部分工作都由 SQL 语句完成。

下面的语句从表中选取 LastName 列的数据：

```
SELECT LastName FROM Persons
```

结果集类似这样：

LastName
Adams
Bush
Carter

在本教程中，我们将为您讲解各种不同的 SQL 语句。

重要事项

一定要记住，*SQL* 对大小写不敏感！

SQL 语句后面的分号？

某些数据库系统要求在每条 SQL 命令的末端使用分号。在我们的教程中不使用分号。

分号是在数据库系统中分隔每条 SQL 语句的标准方法，这样就可以在对服务器的相同请求中执行一条以上的语句。

如果您使用的是 MS Access 和 SQL Server 2000，则不必在每条 SQL 语句之后使用分号，不过某些数据库软件要求必须使用分号。

SQL DML 和 DDL

可以把 SQL 分为两个部分：数据操作语言 (DML) 和 数据定义语言 (DDL)。

SQL (结构化查询语言)是用于执行查询的语法。但是 SQL 语言也包含用于更新、插入和删除记录的语法。

查询和更新指令构成了 SQL 的 DML 部分：

- *SELECT* - 从数据库表中获取数据
- *UPDATE* - 更新数据库表中的数据
- *DELETE* - 从数据库表中删除数据
- *INSERT INTO* - 向数据库表中插入数据

SQL 的数据定义语言 (DDL) 部分使我们有能力创建或删除表格。我们也可以定义索引（键），规定表之间的链接，以及施加表间的约束。

SQL 中最重要的 DDL 语句:

- *CREATE DATABASE* - 创建新数据库
- *ALTER DATABASE* - 修改数据库
- *CREATE TABLE* - 创建新表
- *ALTER TABLE* - 变更（改变）数据库表
- *DROP TABLE* - 删除表
- *CREATE INDEX* - 创建索引（搜索键）
- *DROP INDEX* - 删除索引

SQL SELECT 语句

本章讲解 **SELECT** 和 **SELECT *** 语句。

SQL SELECT 语句

SELECT 语句用于从表中选取数据。

结果被存储在一个结果表中（称为结果集）。

SQL SELECT 语法

```
SELECT 列名称 FROM 表名称
```

以及：

```
SELECT * FROM 表名称
```

注释：SQL 语句对大小写不敏感。SELECT 等效于 select。

SQL SELECT 实例

如需获取名为 "LastName" 和 "FirstName" 的列的内容（从名为 "Persons" 的数据库表），请使用类似这样的 SELECT 语句：

```
SELECT LastName,FirstName FROM Persons
```

"Persons" 表:

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

结果：

LastName	FirstName
Adams	John
Bush	George
Carter	Thomas

SQL SELECT * 实例

现在我们希望从 "Persons" 表中选取所有的列。

请使用符号 * 取代列的名称，就像这样：

```
SELECT * FROM Persons
```

提示：星号（*）是选取所有列的快捷方式。

结果：

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

在结果集（result-set）中导航

由 SQL 查询程序获得的结果被存放在一个结果集中。大多数数据库软件系统都允许使用编程函数在结果集中进行导航，比如：Move-To-First-Record、Get-Record-Content、Move-To-Next-Record 等等。

类似这些编程函数不在本教程讲解之列。如需学习通过函数调用访问数据的知识，请访问我们的 ADO 教程 和 PHP 教程。

SQL SELECT DISTINCT 语句

本章讲解 **SELECT DISTINCT** 语句。

SQL SELECT DISTINCT 语句

在表中，可能会包含重复值。这并不成问题，不过，有时您也许希望仅仅列出不同（distinct）的值。

关键词 DISTINCT 用于返回唯一不同的值。

语法：

```
SELECT DISTINCT 列名称 FROM 表名称
```

使用 DISTINCT 关键词

如果要从 "Company" 列中选取所有的值，我们需要使用 SELECT 语句：

```
SELECT Company FROM Orders
```

"Orders"表：

Company	OrderNumber
IBM	3532
W3School	2356
Apple	4698
W3School	6953

结果：

Company
IBM
W3School
Apple
W3School

请注意，在结果集中，W3School 被列出了两次。

如需从 Company" 列中仅选取唯一不同的值，我们需要使用 SELECT DISTINCT 语句：

```
SELECT DISTINCT Company FROM Orders
```

结果：

Company
IBM
W3School
Apple

现在，在结果集中，"W3School" 仅被列出了一次。

SQL WHERE 子句

WHERE 子句用于规定选择的标准。

WHERE 子句

如需有条件地从表中选取数据，可将 WHERE 子句添加到 SELECT 语句。

语法

```
SELECT 列名称 FROM 表名称 WHERE 列 运算符 值
```

下面的运算符可在 WHERE 子句中使用：

操作符	描述
=	等于
<>	不等于
>	大于
<	小于
>=	大于等于
<=	小于等于
BETWEEN	在某个范围内
LIKE	搜索某种模式

注释：在某些版本的 SQL 中，操作符 <> 可以写为 !=。

使用 WHERE 子句

如果只希望选取居住在城市 "Beijing" 中的人，我们需要向 SELECT 语句添加 WHERE 子句：

```
SELECT * FROM Persons WHERE City='Beijing'
```

"Persons" 表

LastName	FirstName	Address	City	Year
Adams	John	Oxford Street	London	1970
Bush	George	Fifth Avenue	New York	1975
Carter	Thomas	Changan Street	Beijing	1980
Gates	Bill	Xuanwumen 10	Beijing	1985

结果：

LastName	FirstName	Address	City	Year
Carter	Thomas	Changan Street	Beijing	1980
Gates	Bill	Xuanwumen 10	Beijing	1985

引号的使用

请注意，我们在例子中的条件值周围使用的是单引号。

SQL 使用单引号来环绕文本值（大部分数据库系统也接受双引号）。如果是数值，请不要使用引号。

文本值：

```
这是正确的：
SELECT * FROM Persons WHERE FirstName='Bush'

这是错误的：
SELECT * FROM Persons WHERE FirstName=Bush
```

数值：

```
这是正确的：
SELECT * FROM Persons WHERE Year>1965

这是错误的：
SELECT * FROM Persons WHERE Year>'1965'
```

SQL AND & OR 运算符

AND 和 OR 运算符用于基于一个以上的条件对记录进行过滤。

AND 和 OR 运算符

AND 和 OR 可在 WHERE 子语句中把两个或多个条件结合起来。

如果第一个条件和第二个条件都成立，则 AND 运算符显示一条记录。

如果第一个条件和第二个条件中只要有一个成立，则 OR 运算符显示一条记录。

原始的表 (用在例子中的)：

LastName	FirstName	Address	City
Adams	John	Oxford Street	London
Bush	George	Fifth Avenue	New York
Carter	Thomas	Changan Street	Beijing
Carter	William	Xuanwumen 10	Beijing

AND 运算符实例

使用 AND 来显示所有姓为 "Carter" 并且名为 "Thomas" 的人：

```
SELECT * FROM Persons WHERE FirstName='Thomas' AND LastName='Carte
```

结果：

LastName	FirstName	Address	City
Carter	Thomas	Changan Street	Beijing

OR 运算符实例

使用 OR 来显示所有姓为 "Carter" 或者名为 "Thomas" 的人：

```
SELECT * FROM Persons WHERE firstname='Thomas' OR lastname='Carter
```

结果：

LastName	FirstName	Address	City
Carter	Thomas	Changan Street	Beijing
Carter	William	Xuanwumen 10	Beijing

结合 AND 和 OR 运算符

我们也可以把 AND 和 OR 结合起来（使用圆括号来组成复杂的表达式）：

```
SELECT * FROM Persons WHERE (FirstName='Thomas' OR FirstName='Will:
AND LastName='Carter'
```

结果：

LastName	FirstName	Address	City
Carter	Thomas	Changan Street	Beijing
Carter	William	Xuanwumen 10	Beijing

SQL ORDER BY 子句

ORDER BY 语句用于对结果集进行排序。

ORDER BY 语句

ORDER BY 语句用于根据指定的列对结果集进行排序。

ORDER BY 语句默认按照升序对记录进行排序。

如果您希望按照降序对记录进行排序，可以使用 DESC 关键字。

原始的表 (用在例子中的)：

Orders 表:

Company	OrderNumber
IBM	3532
W3School	2356
Apple	4698
W3School	6953

实例 1

以字母顺序显示公司名称：

```
SELECT Company, OrderNumber FROM Orders ORDER BY Company
```

结果：

Company	OrderNumber
Apple	4698
IBM	3532
W3School	6953
W3School	2356

实例 2

以字母顺序显示公司名称（Company），并以数字顺序显示顺序号（OrderNumber）：

```
SELECT Company, OrderNumber FROM Orders ORDER BY Company, OrderNumb
```

结果：

Company	OrderNumber
Apple	4698
IBM	3532
W3School	2356
W3School	6953

实例 3

以逆字母顺序显示公司名称：

```
SELECT Company, OrderNumber FROM Orders ORDER BY Company DESC
```

结果：

Company	OrderNumber
W3School	6953
W3School	2356
IBM	3532
Apple	4698

实例 4

以逆字母顺序显示公司名称，并以数字顺序显示顺序号：

```
SELECT Company, OrderNumber FROM Orders ORDER BY Company DESC, Orde
```

结果：

Company	OrderNumber
W3School	2356
W3School	6953
IBM	3532
Apple	4698

注意：在以上的结果中有两个相等的公司名称 (W3School)。只有这一次，在第一列中有相同的值时，第二列是以升序排列的。如果第一列中有些值为 nulls 时，情况也是这样的。

SQL INSERT INTO 语句

INSERT INTO 语句

INSERT INTO 语句用于向表格中插入新的行。

语法

```
INSERT INTO 表名称 VALUES (值1, 值2,....)
```

我们也可以指定所要插入数据的列：

```
INSERT INTO table_name (列1, 列2,...) VALUES (值1, 值2,....)
```

插入新的行

"Persons" 表：

LastName	FirstName	Address	City
Carter	Thomas	Changan Street	Beijing

SQL 语句：

```
INSERT INTO Persons VALUES ('Gates', 'Bill', 'Xuanwumen 10', 'Beij
```

结果：

LastName	FirstName	Address	City
Carter	Thomas	Changan Street	Beijing
Gates	Bill	Xuanwumen 10	Beijing

在指定的列中插入数据

"Persons" 表：

LastName	FirstName	Address	City
Carter	Thomas	Changan Street	Beijing
Gates	Bill	Xuanwumen 10	Beijing

SQL 语句：

```
INSERT INTO Persons (LastName, Address) VALUES ('Wilson', 'Champs-E
```

结果：

LastName	FirstName	Address	City
Carter	Thomas	Changan Street	Beijing
Gates	Bill	Xuanwumen 10	Beijing
Wilson	Champs-Elysees		

SQL UPDATE 语句

Update 语句

Update 语句用于修改表中的数据。

语法：

```
UPDATE 表名称 SET 列名称 = 新值 WHERE 列名称 = 某值
```

Person:

LastName	FirstName	Address	City
Gates	Bill	Xuanwumen 10	Beijing
Wilson	Champs-Elysees		

更新某一行中的一个列

我们为 lastname 是 "Wilson" 的人添加 firstname：

```
UPDATE Person SET FirstName = 'Fred' WHERE LastName = 'Wilson'
```

结果：

LastName	FirstName	Address	City
Gates	Bill	Xuanwumen 10	Beijing
Wilson	Fred	Champs-Elysees	

更新某一行中的若干列

我们会修改地址（address），并添加城市名称（city）：

```
UPDATE Person SET Address = 'Zhongshan 23', City = 'Nanjing'
WHERE LastName = 'Wilson'
```

结果：

LastName	FirstName	Address	City
Gates	Bill	Xuanwumen 10	Beijing
Wilson	Fred	Zhongshan 23	Nanjing

SQL DELETE 语句

DELETE 语句

DELETE 语句用于删除表中的行。

语法

```
DELETE FROM 表名称 WHERE 列名称 = 值
```

Person:

LastName	FirstName	Address	City
Gates	Bill	Xuanwumen 10	Beijing
Wilson	Fred	Zhongshan 23	Nanjing

删除某行

"Fred Wilson" 会被删除：

```
DELETE FROM Person WHERE LastName = 'Wilson'
```

结果:

LastName	FirstName	Address	City
Gates	Bill	Xuanwumen 10	Beijing

删除所有行

可以在不删除表的情况下删除所有的行。这意味着表的结构、属性和索引都是完整的：

```
DELETE FROM table_name
```

或者：

```
DELETE * FROM table_name
```

SQL高级

SQL TOP 子句

TOP 子句

TOP 子句用于规定要返回的记录的数目。

对于拥有数千条记录的大型表来说，TOP 子句是非常有用的。

注释：并非所有的数据库系统都支持 TOP 子句。

SQL Server 的语法：

```
SELECT TOP number|percent column_name(s)
FROM table_name
```

MySQL 和 Oracle 中的 SQL SELECT TOP 是等价的

MySQL 语法

```
SELECT column_name(s)
FROM table_name
LIMIT number
```

例子

```
SELECT *
FROM Persons
LIMIT 5
```

Oracle 语法

```
SELECT column_name(s)
FROM table_name
WHERE ROWNUM <= number
```

例子

```
SELECT *
FROM Persons
WHERE ROWNUM <= 5
```

原始的表 (用在例子中的)：

Persons 表:

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing
4	Obama	Barack	Pennsylvania Avenue	Washington

SQL TOP 实例

现在，我们希望从上面的 "Persons" 表中选取头两条记录。

我们可以使用下面的 SELECT 语句：

```
SELECT TOP 2 * FROM Persons
```

结果：

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York

SQL TOP PERCENT 实例

现在，我们希望从上面的 "Persons" 表中选取 50% 的记录。

我们可以使用下面的 SELECT 语句：

```
SELECT TOP 50 PERCENT * FROM Persons
```

结果：

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York

SQL LIKE 操作符

LIKE 操作符用于在 **WHERE** 子句中搜索列中的指定模式。

LIKE 操作符

LIKE 操作符用于在 WHERE 子句中搜索列中的指定模式。

SQL LIKE 操作符语法

```
SELECT column_name(s)
FROM table_name
WHERE column_name LIKE pattern
```

原始的表 (用在例子中的)：

Persons 表:

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

LIKE 操作符实例

例子 1

现在，我们希望从上面的 "Persons" 表中选取居住在以 "N" 开始的城市里的人：

我们可以使用下面的 SELECT 语句：

```
SELECT * FROM Persons
WHERE City LIKE 'N%'
```

提示："%" 可用于定义通配符（模式中缺少的字母）。

结果集：

Id	LastName	FirstName	Address	City
2	Bush	George	Fifth Avenue	New York

例子 2

接下来，我们希望从 "Persons" 表中选取居住在以 "g" 结尾的城市里的人：

我们可以使用下面的 SELECT 语句：

```
SELECT * FROM Persons
WHERE City LIKE '%g'
```

结果集：

Id	LastName	FirstName	Address	City
3	Carter	Thomas	Changan Street	Beijing

例子 3

接下来，我们希望从 "Persons" 表中选取居住在包含 "lon" 的城市里的人：

我们可以使用下面的 SELECT 语句：

```
SELECT * FROM Persons
WHERE City LIKE '%lon%'
```

结果集：

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London

例子 4

通过使用 NOT 关键字，我们可以从 "Persons" 表中选取居住在不包含 "lon" 的城市里的人：

我们可以使用下面的 SELECT 语句：

```
SELECT * FROM Persons
WHERE City NOT LIKE '%lon%'
```

结果集：

Id	LastName	FirstName	Address	City
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

SQL 通配符

在搜索数据库中的数据时，您可以使用 **SQL** 通配符。

SQL 通配符

在搜索数据库中的数据时，SQL 通配符可以替代一个或多个字符。

SQL 通配符必须与 LIKE 运算符一起使用。

在 SQL 中，可使用以下通配符：

通配符	描述
%	替代一个或多个字符
_	仅替代一个字符
[charlist]	字符列中的任何单一字符
[^charlist] 或者 [!charlist]	不在字符列中的任何单一字符

原始的表 (用在例子中的)： **Persons** 表:

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

使用 % 通配符

例子 1

现在，我们希望从上面的 "Persons" 表中选取居住在以 "Ne" 开始的城市里的人：

我们可以使用下面的 SELECT 语句：

```
SELECT * FROM Persons
WHERE City LIKE 'Ne%'
```

结果集：

Id	LastName	FirstName	Address	City
2	Bush	George	Fifth Avenue	New York

例子 2

接下来，我们希望从 "Persons" 表中选取居住在包含 "lond" 的城市里的人：

我们可以使用下面的 SELECT 语句：

```
SELECT * FROM Persons
WHERE City LIKE '%lond%'
```

结果集：

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London

使用 _ 通配符

例子 1

现在，我们希望从上面的 "Persons" 表中选取名字的第一个字符之后是 "eorge" 的人：

我们可以使用下面的 SELECT 语句：

```
SELECT * FROM Persons
WHERE FirstName LIKE '_eorge'
```

结果集：

Id	LastName	FirstName	Address	City
2	Bush	George	Fifth Avenue	New York

例子 2

接下来，我们希望从 "Persons" 表中选取的这条记录的姓氏以 "C" 开头，然后是一个任意字符，然后是 "r"，然后是任意字符，然后是 "er"：

我们可以使用下面的 SELECT 语句：

```
SELECT * FROM Persons
WHERE LastName LIKE 'C_r_er'
```

结果集：

Id	LastName	FirstName	Address	City
3	Carter	Thomas	Changan Street	Beijing

使用 [charlist] 通配符

例子 1

现在，我们希望从上面的 "Persons" 表中选取居住的城市以 "A" 或 "L" 或 "N" 开头的人：

我们可以使用下面的 SELECT 语句：

```
SELECT * FROM Persons
WHERE City LIKE '[ALN]%'
```

结果集：

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York

例子 2

现在，我们希望从上面的 "Persons" 表中选取居住的城市不以 "A" 或 "L" 或 "N" 开头的人：

我们可以使用下面的 SELECT 语句：

```
SELECT * FROM Persons
WHERE City LIKE '[!ALN]%'
```

结果集：

Id	LastName	FirstName	Address	City
3	Carter	Thomas	Changan Street	Beijing

SQL IN 操作符

IN 操作符

IN 操作符允许我们在 WHERE 子句中规定多个值。

SQL IN 语法

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (value1,value2,...)
```

原始的表 (在实例中使用：)

Persons 表:

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

IN 操作符实例

现在，我们希望从上表中选取姓氏为 Adams 和 Carter 的人：

我们可以使用下面的 SELECT 语句：

```
SELECT * FROM Persons
WHERE LastName IN ('Adams','Carter')
```

结果集：

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
3	Carter	Thomas	Changan Street	Beijing

SQL BETWEEN 操作符

BETWEEN 操作符在 **WHERE** 子句中使用，作用是选取介于两个值之间的数据范围。

BETWEEN 操作符

操作符 BETWEEN ... AND 会选取介于两个值之间的数据范围。这些值可以是数值、文本或者日期。

SQL BETWEEN 语法

```
SELECT column_name(s)
FROM table_name
WHERE column_name
BETWEEN value1 AND value2
```

原始的表 (在实例中使用：)

Persons 表:

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing
4	Gates	Bill	Xuanwumen 10	Beijing

BETWEEN 操作符实例

如需以字母顺序显示介于 "Adams"（包括）和 "Carter"（不包括）之间的人，请使用下面的 SQL：

```
SELECT * FROM Persons
WHERE LastName
BETWEEN 'Adams' AND 'Carter'
```

结果集：

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York

重要事项：不同的数据库对 BETWEEN...AND 操作符的处理方式是有差异的。某些数据库会列出介于 "Adams" 和 "Carter" 之间的人，但不包括 "Adams" 和 "Carter" ；某些数据库会列出介于 "Adams" 和 "Carter" 之间并包括 "Adams" 和 "Carter" 的人；而另一些数据库会列出介于 "Adams" 和 "Carter" 之间的人，包括 "Adams" ，但不包括 "Carter" 。

所以，请检查你的数据库是如何处理 BETWEEN....AND 操作符的！

实例 2

如需使用上面的例子显示范围之外的人，请使用 NOT 操作符：

```
SELECT * FROM Persons
WHERE LastName
NOT BETWEEN 'Adams' AND 'Carter'
```

结果集：

Id	LastName	FirstName	Address	City
3	Carter	Thomas	Changan Street	Beijing
4	Gates	Bill	Xuanwumen 10	Beijing

SQL Alias（别名）

通过使用 **SQL**，可以为列名称和表名称指定别名（**Alias**）。

SQL Alias

表的 SQL Alias 语法

```
SELECT column_name(s)
FROM table_name
AS alias_name
```

列的 SQL Alias 语法

```
SELECT column_name AS alias_name
FROM table_name
```

Alias 实例: 使用表名称别名

假设我们有两个表分别是："Persons" 和 "Product_Orders"。我们分别为它们指定别名 "p" 和 "po"。

现在，我们希望列出 "John Adams" 的所有定单。

我们可以使用下面的 SELECT 语句：

```
SELECT po.OrderID, p.LastName, p.FirstName
FROM Persons AS p, Product_Orders AS po
WHERE p.LastName='Adams' AND p.FirstName='John'
```

不使用别名的 SELECT 语句：

```
SELECT Product_Orders.OrderID, Persons.LastName, Persons.FirstName
FROM Persons, Product_Orders
WHERE Persons.LastName='Adams' AND Persons.FirstName='John'
```

从上面两条 SELECT 语句您可以看到，别名使查询程序更易阅读和书写。

Alias 实例: 使用一个列名别名

表 Persons:

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

SQL:

```
SELECT LastName AS Family, FirstName AS Name
FROM Persons
```

结果：

Family	Name
Adams	John
Bush	George
Carter	Thomas

SQL JOIN

SQL join 用于根据两个或多个表中的列之间的关系，从这些表中查询数据。

Join 和 Key

有时为了得到完整的结果，我们需要从两个或更多的表中获取结果。我们就需要执行 join。

数据库中的表可通过键将彼此联系起来。主键（Primary Key）是一个列，在这个列中的每一行的值都是唯一的。在表中，每个主键的值都是唯一的。这样做的目的是在不重复每个表中的所有数据的情况下，把表间的数据交叉捆绑在一起。

请看 "Persons" 表：

Id_P	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

请注意，"Id_P" 列是 Persons 表中的的主键。这意味着没有两行能够拥有相同的 Id_P。即使两个人的姓名完全相同，Id_P 也可以区分他们。

接下来请看 "Orders" 表：

Id_O	OrderNo	Id_P
1	77895	3
2	44678	3
3	22456	1
4	24562	1
5	34764	65

请注意，"Id_O" 列是 Orders 表中的的主键，同时，"Orders" 表中的 "Id_P" 列用于引用 "Persons" 表中的人，而无需使用他们的确切姓名。

请留意，"Id_P" 列把上面的两个表联系了起来。

引用两个表

我们可以通过引用两个表的方式，从两个表中获取数据：

谁订购了产品，并且他们订购了什么产品？

```
SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo
FROM Persons, Orders
WHERE Persons.Id_P = Orders.Id_P
```

结果集：

LastName	FirstName	OrderNo
Adams	John	22456
Adams	John	24562
Carter	Thomas	77895
Carter	Thomas	44678

SQL JOIN - 使用 Join

除了上面的方法，我们也可以使用关键词 JOIN 来从两个表中获取数据。

如果我们希望列出所有人的定购，可以使用下面的 SELECT 语句：

```
SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo
FROM Persons
INNER JOIN Orders
ON Persons.Id_P = Orders.Id_P
ORDER BY Persons.LastName
```

结果集：

LastName	FirstName	OrderNo
Adams	John	22456
Adams	John	24562
Carter	Thomas	77895
Carter	Thomas	44678

不同的 SQL JOIN

除了我们在上面的例子中使用的 INNER JOIN（内连接），我们还可以使用其他几种连接。

下面列出了您可以使用的 JOIN 类型，以及它们之间的差异。

- JOIN: 如果表中有至少一个匹配，则返回行
- LEFT JOIN: 即使右表中没有匹配，也从左表返回所有的行
- RIGHT JOIN: 即使左表中没有匹配，也从右表返回所有的行
- FULL JOIN: 只要其中一个表中存在匹配，就返回行

SQL INNER JOIN 关键字

SQL INNER JOIN 关键字

在表中存在至少一个匹配时，INNER JOIN 关键字返回行。

INNER JOIN 关键字语法

```
SELECT column_name(s)
FROM table_name1
INNER JOIN table_name2
ON table_name1.column_name=table_name2.column_name
```

注释：INNER JOIN 与 JOIN 是相同的。

原始的表 (用在例子中的)：

"Persons" 表：

Id_P	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

"Orders" 表：

Id_O	OrderNo	Id_P
1	77895	3
2	44678	3
3	22456	1
4	24562	1
5	34764	65

内连接（INNER JOIN）实例

现在，我们希望列出所有人的定购。

您可以使用下面的 SELECT 语句：

```
SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo
FROM Persons
INNER JOIN Orders
ON Persons.Id_P=Orders.Id_P
ORDER BY Persons.LastName
```

结果集：

LastName	FirstName	OrderNo
Adams	John	22456
Adams	John	24562
Carter	Thomas	77895
Carter	Thomas	44678

INNER JOIN 关键字在表中存在至少一个匹配时返回行。如果 "Persons" 中的行在 "Orders" 中没有匹配，就不会列出这些行。

SQL LEFT JOIN 关键字

SQL LEFT JOIN 关键字

LEFT JOIN 关键字会从左表 (table_name1) 那里返回所有的行，即使在右表 (table_name2) 中没有匹配的行。

LEFT JOIN 关键字语法

```
SELECT column_name(s)
FROM table_name1
LEFT JOIN table_name2
ON table_name1.column_name=table_name2.column_name
```

注释：在某些数据库中， LEFT JOIN 称为 LEFT OUTER JOIN。

原始的表 (用在例子中的)：

"Persons" 表：

Id_P	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

"Orders" 表：

Id_O	OrderNo	Id_P
1	77895	3
2	44678	3
3	22456	1
4	24562	1
5	34764	65

左连接（LEFT JOIN）实例

现在，我们希望列出所有的人，以及他们的定购 - 如果有的话。

您可以使用下面的 SELECT 语句：

```
SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo
FROM Persons
LEFT JOIN Orders
ON Persons.Id_P=Orders.Id_P
ORDER BY Persons.LastName
```

结果集：

LastName	FirstName	OrderNo
Adams	John	22456
Adams	John	24562
Carter	Thomas	77895
Carter	Thomas	44678
Bush	George	

LEFT JOIN 关键字会从左表 (Persons) 那里返回所有的行，即使在右表 (Orders) 中没有匹配的行。

SQL RIGHT JOIN 关键字

SQL RIGHT JOIN 关键字

RIGHT JOIN 关键字会右表 (table_name2) 那里返回所有的行，即使在左表 (table_name1) 中没有匹配的行。

RIGHT JOIN 关键字语法

```
SELECT column_name(s)
FROM table_name1
RIGHT JOIN table_name2
ON table_name1.column_name=table_name2.column_name
```

注释：在某些数据库中， RIGHT JOIN 称为 RIGHT OUTER JOIN。

原始的表 (用在例子中的)：

"Persons" 表：

Id_P	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

"Orders" 表：

Id_O	OrderNo	Id_P
1	77895	3
2	44678	3
3	22456	1
4	24562	1
5	34764	65

右连接（RIGHT JOIN）实例

现在，我们希望列出所有的定单，以及定购它们的人 - 如果有的话。

您可以使用下面的 SELECT 语句：

```
SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo
FROM Persons
RIGHT JOIN Orders
ON Persons.Id_P=Orders.Id_P
ORDER BY Persons.LastName
```

结果集：

LastName	FirstName	OrderNo
Adams	John	22456
Adams	John	24562
Carter	Thomas	77895
Carter	Thomas	44678
34764		

RIGHT JOIN 关键字会从右表 (Orders) 那里返回所有的行，即使在左表 (Persons) 中没有匹配的行。

SQL FULL JOIN 关键字

SQL FULL JOIN 关键字

只要其中某个表存在匹配，FULL JOIN 关键字就会返回行。

FULL JOIN 关键字语法

```
SELECT column_name(s)
FROM table_name1
FULL JOIN table_name2
ON table_name1.column_name=table_name2.column_name
```

注释：在某些数据库中， FULL JOIN 称为 FULL OUTER JOIN。

原始的表 (用在例子中的)：

"Persons" 表：

Id_P	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

"Orders" 表：

Id_O	OrderNo	Id_P
1	77895	3
2	44678	3
3	22456	1
4	24562	1
5	34764	65

全连接（FULL JOIN）实例

现在，我们希望列出所有的人，以及他们的定单，以及所有的定单，以及定购它们的人。

您可以使用下面的 SELECT 语句：

```
SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo
FROM Persons
FULL JOIN Orders
ON Persons.Id_P=Orders.Id_P
ORDER BY Persons.LastName
```

结果集：

LastName	FirstName	OrderNo
Adams	John	22456
Adams	John	24562
Carter	Thomas	77895
Carter	Thomas	44678
Bush	George	
34764		

FULL JOIN 关键字会从左表 (Persons) 和右表 (Orders) 那里返回所有的行。如果 "Persons" 中的行在表 "Orders" 中没有匹配，或者如果 "Orders" 中的行在表 "Persons" 中没有匹配，这些行同样会列出。

SQL UNION 和 UNION ALL 操作符

SQL UNION 操作符

UNION 操作符用于合并两个或多个 SELECT 语句的结果集。

请注意，UNION 内部的 SELECT 语句必须拥有相同数量的列。列也必须拥有相似的数据类型。同时，每条 SELECT 语句中的列的顺序必须相同。

SQL UNION 语法

```
SELECT column_name(s) FROM table_name1
UNION
SELECT column_name(s) FROM table_name2
```

注释：默认地，UNION 操作符选取不同的值。如果允许重复的值，请使用 UNION ALL。

SQL UNION ALL 语法

```
SELECT column_name(s) FROM table_name1
UNION ALL
SELECT column_name(s) FROM table_name2
```

另外，UNION 结果集中的列名总是等于 UNION 中第一个 SELECT 语句中的列名。

下面的例子中使用的原始表：

Employees_China:

E_ID	E_Name
01	Zhang, Hua
02	Wang, Wei
03	Carter, Thomas
04	Yang, Ming

Employees_USA:

E_ID	E_Name
01	Adams, John
02	Bush, George
03	Carter, Thomas
04	Gates, Bill

使用 UNION 命令

实例

列出所有在中国和美国的不同的雇员名：

```
SELECT E_Name FROM Employees_China
UNION
SELECT E_Name FROM Employees_USA
```

结果

E_Name
Zhang, Hua
Wang, Wei
Carter, Thomas
Yang, Ming
Adams, John
Bush, George
Gates, Bill

注释：这个命令无法列出在中国和美国的所有雇员。在上面的例子中，我们有两个名字相同的雇员，他们当中只有一个人被列出来了。UNION 命令只会选取不同的值。

UNION ALL

UNION ALL 命令和 UNION 命令几乎是等效的，不过 UNION ALL 命令会列出所有的值。

```
SQL Statement 1
UNION ALL
SQL Statement 2
```

使用 UNION ALL 命令

实例：

列出在中国和美国的所有的雇员：

```
SELECT E_Name FROM Employees_China
UNION ALL
SELECT E_Name FROM Employees_USA
```

结果

E_Name
Zhang, Hua
Wang, Wei
Carter, Thomas
Yang, Ming
Adams, John
Bush, George
Carter, Thomas
Gates, Bill

SQL SELECT INTO 语句

SQL SELECT INTO 语句可用于创建表的备份复件。

SELECT INTO 语句

SELECT INTO 语句从一个表中选取数据，然后把数据插入另一个表中。

SELECT INTO 语句常用于创建表的备份复件或者用于对记录进行存档。

SQL SELECT INTO 语法

您可以把所有的列插入新表：

```
SELECT *
INTO new_table_name [IN externaldatabase]
FROM old_tablename
```

或者只把希望的列插入新表：

```
SELECT column_name(s)
INTO new_table_name [IN externaldatabase]
FROM old_tablename
```

SQL SELECT INTO 实例 - 制作备份复件

下面的例子会制作 "Persons" 表的备份复件：

```
SELECT *
INTO Persons_backup
FROM Persons
```

IN 子句可用于向另一个数据库中拷贝表：

```
SELECT *
INTO Persons IN 'Backup.mdb'
FROM Persons
```

如果我们希望拷贝某些域，可以在 SELECT 语句后列出这些域：

```
SELECT LastName,FirstName
INTO Persons_backup
FROM Persons
```

SQL SELECT INTO 实例 - 带有 WHERE 子句

我们也可以添加 WHERE 子句。

下面的例子通过从 "Persons" 表中提取居住在 "Beijing" 的人的信息，创建了一个带有两个列的名为 "Persons_backup" 的表：

```
SELECT LastName,Firstname
INTO Persons_backup
FROM Persons
WHERE City='Beijing'
```

SQL SELECT INTO 实例 - 被连接的表

从一个以上的表中选取数据也是可以做到的。

下面的例子会创建一个名为 "Persons_Order_Backup" 的新表，其中包含了从 Persons 和 Orders 两个表中取得的信息：

```
SELECT Persons.LastName,Orders.OrderNo
INTO Persons_Order_Backup
FROM Persons
INNER JOIN Orders
ON Persons.Id_P=Orders.Id_P
```

SQL CREATE DATABASE 语句

CREATE DATABASE 语句

CREATE DATABASE 用于创建数据库。

SQL CREATE DATABASE 语法

```
CREATE DATABASE database_name
```

SQL CREATE DATABASE 实例

现在我们希望创建一个名为 "my_db" 的数据库。

我们使用下面的 CREATE DATABASE 语句：

```
CREATE DATABASE my_db
```

可以通过 CREATE TABLE 来添加数据库表。

SQL CREATE TABLE 语句

CREATE TABLE 语句

CREATE TABLE 语句用于创建数据库中的表。

SQL CREATE TABLE 语法

```
CREATE TABLE 表名称
(
列名称1 数据类型,
列名称2 数据类型,
列名称3 数据类型,
....
)
```

数据类型（data_type）规定了列可容纳何种数据类型。下面的表格包含了SQL中最常用的数据类型：

数据类型	描述
integer(size) int(size) smallint(size) tinyint(size)	仅容纳整数。在括号内规定数字的最大位数。
decimal(size,d) numeric(size,d)	容纳带有小数的数字。 "size" 规定数字的最大位数。"d" 规定小数点右侧的最大位数。
char(size)	容纳固定长度的字符串（可容纳字母、数字以及特殊字符）。 在括号中规定字符串的长度。
varchar(size)	容纳可变长度的字符串（可容纳字母、数字以及特殊的字符）。 在括号中规定字符串的最大长度。
date(yyyymmdd)	容纳日期。

SQL CREATE TABLE 实例

本例演示如何创建名为 "Person" 的表。

该表包含 5 个列，列名分别是："Id_P"、"LastName"、"FirstName"、"Address" 以及 "City"：

```
CREATE TABLE Persons
(
Id_P int,
LastName varchar(255),
FirstName varchar(255),
Address varchar(255),
City varchar(255)
)
```

Id_P 列的数据类型是 int，包含整数。其余 4 列的数据类型是 varchar，最大长度为 255 个字符。

空的 "Persons" 表类似这样：

Id_P	LastName	FirstName	Address	City

可使用 INSERT INTO 语句向空表写入数据。

SQL 约束 (Constraints)

SQL 约束

约束用于限制加入表的数据的类型。

可以在创建表时规定约束（通过 CREATE TABLE 语句），或者在表创建之后也可以（通过 ALTER TABLE 语句）。

我们将主要探讨以下几种约束：

- NOT NULL
- UNIQUE
- PRIMARY KEY
- FOREIGN KEY
- CHECK
- DEFAULT

注释：在下面的章节，我们会详细讲解每一种约束。

SQL NOT NULL 约束

SQL NOT NULL 约束

NOT NULL 约束强制列不接受 NULL 值。

NOT NULL 约束强制字段始终包含值。这意味着，如果不向字段添加值，就无法插入新记录或者更新记录。

下面的 SQL 语句强制 "Id_P" 列和 "LastName" 列不接受 NULL 值：

```
CREATE TABLE Persons
(
Id_P int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255)
)
```

SQL UNIQUE 约束

SQL UNIQUE 约束

UNIQUE 约束唯一标识数据库表中的每条记录。

UNIQUE 和 PRIMARY KEY 约束均为列或列集合提供了唯一性的保证。

PRIMARY KEY 拥有自动定义的 UNIQUE 约束。

请注意，每个表可以有多个 UNIQUE 约束，但是每个表只能有一个 PRIMARY KEY 约束。

SQL UNIQUE Constraint on CREATE TABLE

下面的 SQL 在 "Persons" 表创建时在 "Id_P" 列创建 UNIQUE 约束：

MySQL:

```
CREATE TABLE Persons
(
Id_P int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255),
UNIQUE (Id_P)
)
```

SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons
(
Id_P int NOT NULL UNIQUE,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255)
)
```

如果需要命名 UNIQUE 约束，以及为多个列定义 UNIQUE 约束，请使用下面的 SQL 语法：

MySQL / SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons
(
Id_P int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255),
CONSTRAINT uc_PersonID UNIQUE (Id_P,LastName)
)
```

SQL UNIQUE Constraint on ALTER TABLE

当表已被创建时，如需在 "Id_P" 列创建 UNIQUE 约束，请使用下列 SQL：

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
ADD UNIQUE (Id_P)
```

如需命名 UNIQUE 约束，并定义多个列的 UNIQUE 约束，请使用下面的 SQL 语法：

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
ADD CONSTRAINT uc_PersonID UNIQUE (Id_P,LastName)
```

撤销 UNIQUE 约束

如需撤销 UNIQUE 约束，请使用下面的 SQL：

MySQL:

```
ALTER TABLE Persons
DROP INDEX uc_PersonID
```

SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
DROP CONSTRAINT uc_PersonID
```

SQL PRIMARY KEY 约束

SQL PRIMARY KEY 约束

PRIMARY KEY 约束唯一标识数据库表中的每条记录。

主键必须包含唯一的值。

主键列不能包含 NULL 值。

每个表都应该有一个主键，并且每个表只能有一个主键。

SQL PRIMARY KEY Constraint on CREATE TABLE

下面的 SQL 在 "Persons" 表创建时在 "Id_P" 列创建 PRIMARY KEY 约束：

MySQL:

```
CREATE TABLE Persons
(
Id_P int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255),
PRIMARY KEY (Id_P)
)
```

SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons
(
Id_P int NOT NULL PRIMARY KEY,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255)
)
```

如果需要命名 PRIMARY KEY 约束，以及为多个列定义 PRIMARY KEY 约束，请使用下面的 SQL 语法：

MySQL / SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons
(
Id_P int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255),
CONSTRAINT pk_PersonID PRIMARY KEY (Id_P,LastName)
)
```

SQL PRIMARY KEY Constraint on ALTER TABLE

如果在表已存在的情况下为 "Id_P" 列创建 PRIMARY KEY 约束，请使用下面的 SQL：

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
ADD PRIMARY KEY (Id_P)
```

如果需要命名 PRIMARY KEY 约束，以及为多个列定义 PRIMARY KEY 约束，请使用下面的 SQL 语法：

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
ADD CONSTRAINT pk_PersonID PRIMARY KEY (Id_P,LastName)
```

注释：如果您使用 ALTER TABLE 语句添加主键，必须把主键列声明为不包含 NULL 值（在表首次创建时）。

撤销 PRIMARY KEY 约束

如需撤销 PRIMARY KEY 约束，请使用下面的 SQL：

MySQL:

```
ALTER TABLE Persons
DROP PRIMARY KEY
```

SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
DROP CONSTRAINT pk_PersonID
```

SQL FOREIGN KEY 约束

SQL FOREIGN KEY 约束

一个表中的 FOREIGN KEY 指向另一个表中的 PRIMARY KEY。

让我们通过一个例子来解释外键。请看下面两个表：

"Persons" 表：

Id_P	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

"Orders" 表：

Id_O	OrderNo	Id_P
1	77895	3
2	44678	3
3	22456	1
4	24562	1

请注意，"Orders" 中的 "Id_P" 列指向 "Persons" 表中的 "Id_P" 列。

"Persons" 表中的 "Id_P" 列是 "Persons" 表中的 PRIMARY KEY。

"Orders" 表中的 "Id_P" 列是 "Orders" 表中的 FOREIGN KEY。

FOREIGN KEY 约束用于预防破坏表之间连接的动作。

FOREIGN KEY 约束也能防止非法数据插入外键列，因为它必须是它指向的那个表中的值之一。

SQL FOREIGN KEY Constraint on CREATE TABLE

下面的 SQL 在 "Orders" 表创建时为 "Id_P" 列创建 FOREIGN KEY：

MySQL:

```
CREATE TABLE Orders
(
Id_O int NOT NULL,
OrderNo int NOT NULL,
Id_P int,
PRIMARY KEY (Id_O),
FOREIGN KEY (Id_P) REFERENCES Persons(Id_P)
)
```

SQL Server / Oracle / MS Access:

```
CREATE TABLE Orders
(
Id_O int NOT NULL PRIMARY KEY,
OrderNo int NOT NULL,
Id_P int FOREIGN KEY REFERENCES Persons(Id_P)
)
```

如果需要命名 FOREIGN KEY 约束，以及为多个列定义 FOREIGN KEY 约束，请使用下面的 SQL 语法：

MySQL / SQL Server / Oracle / MS Access:

```
CREATE TABLE Orders
(
Id_O int NOT NULL,
OrderNo int NOT NULL,
Id_P int,
PRIMARY KEY (Id_O),
CONSTRAINT fk_PerOrders FOREIGN KEY (Id_P)
REFERENCES Persons(Id_P)
)
```

SQL FOREIGN KEY Constraint on ALTER TABLE

如果在 "Orders" 表已存在的情况下为 "Id_P" 列创建 FOREIGN KEY 约束，请使用下面的 SQL：

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE Orders
ADD FOREIGN KEY (Id_P)
REFERENCES Persons(Id_P)
```

如果需要命名 FOREIGN KEY 约束，以及为多个列定义 FOREIGN KEY 约束，请使用下面的 SQL 语法：

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE Orders
ADD CONSTRAINT fk_PerOrders
FOREIGN KEY (Id_P)
REFERENCES Persons(Id_P)
```

撤销 FOREIGN KEY 约束

如需撤销 FOREIGN KEY 约束，请使用下面的 SQL：

MySQL:

```
ALTER TABLE Orders
DROP FOREIGN KEY fk_PerOrders
```

SQL Server / Oracle / MS Access:

```
ALTER TABLE Orders
DROP CONSTRAINT fk_PerOrders
```

SQL CHECK 约束

SQL CHECK 约束

CHECK 约束用于限制列中的值的范围。

如果对单个列定义 CHECK 约束，那么该列只允许特定的值。

如果对一个表定义 CHECK 约束，那么此约束会在特定的列中对值进行限制。

SQL CHECK Constraint on CREATE TABLE

下面的 SQL 在 "Persons" 表创建时为 "Id_P" 列创建 CHECK 约束。CHECK 约束规定 "Id_P" 列必须只包含大于 0 的整数。

My SQL:

```
CREATE TABLE Persons
(
Id_P int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255),
CHECK (Id_P>0)
)
```

SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons
(
Id_P int NOT NULL CHECK (Id_P>0),
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255)
)
```

如果需要命名 CHECK 约束，以及为多个列定义 CHECK 约束，请使用下面的 SQL 语法：

MySQL / SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons
(
Id_P int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255),
CONSTRAINT chk_Person CHECK (Id_P>0 AND City='Sandnes')
)
```

SQL CHECK Constraint on ALTER TABLE

如果在表已存在的情况下为 "Id_P" 列创建 CHECK 约束，请使用下面的 SQL：

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
ADD CHECK (Id_P>0)
```

如果需要命名 CHECK 约束，以及为多个列定义 CHECK 约束，请使用下面的 SQL 语法：

MySQL / SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
ADD CONSTRAINT chk_Person CHECK (Id_P>0 AND City='Sandnes')
```

撤销 CHECK 约束

如需撤销 CHECK 约束，请使用下面的 SQL：

SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
DROP CONSTRAINT chk_Person
```

MySQL:

```
ALTER TABLE Persons
DROP CHECK chk_Person
```

SQL DEFAULT 约束

SQL DEFAULT 约束

DEFAULT 约束用于向列中插入默认值。

如果没有规定其他的值，那么会将默认值添加到所有的新记录。

SQL DEFAULT Constraint on CREATE TABLE

下面的 SQL 在 "Persons" 表创建时为 "City" 列创建 DEFAULT 约束：

My SQL / SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons
(
Id_P int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255) DEFAULT 'Sandnes'
)
```

通过使用类似 GETDATE() 这样的函数，DEFAULT 约束也可以用于插入系统值：

```
CREATE TABLE Orders
(
Id_O int NOT NULL,
OrderNo int NOT NULL,
Id_P int,
OrderDate date DEFAULT GETDATE()
)
```

SQL DEFAULT Constraint on ALTER TABLE

如果在表已存在的情况下为 "City" 列创建 DEFAULT 约束，请使用下面的 SQL：

MySQL:

```
ALTER TABLE Persons
ALTER City SET DEFAULT 'SANDNES'
```

SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
ALTER COLUMN City SET DEFAULT 'SANDNES'
```

撤销 DEFAULT 约束

如需撤销 DEFAULT 约束，请使用下面的 SQL：

MySQL:

```
ALTER TABLE Persons
ALTER City DROP DEFAULT
```

SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
ALTER COLUMN City DROP DEFAULT
```

SQL CREATE INDEX 语句

CREATE INDEX 语句用于在表中创建索引。

在不读取整个表的情况下，索引使数据库应用程序可以更快地查找数据。

索引

您可以在表中创建索引，以便更加快速高效地查询数据。

用户无法看到索引，它们只能被用来加速搜索/查询。

注释：更新一个包含索引的表需要比更新一个没有索引的表更多的时间，这是由于索引本身也需要更新。因此，理想的做法是仅仅在常常被搜索的列（以及表）上面创建索引。

SQL CREATE INDEX 语法

在表上创建一个简单的索引。允许使用重复的值：

```
CREATE INDEX index_name
ON table_name (column_name)
```

注释："column_name" 规定需要索引的列。

SQL CREATE UNIQUE INDEX 语法

在表上创建一个唯一的索引。唯一的索引意味着两个行不能拥有相同的索引值。

```
CREATE UNIQUE INDEX index_name
ON table_name (column_name)
```

CREATE INDEX 实例

本例会创建一个简单的索引，名为 "PersonIndex"，在 Person 表的 LastName 列：

```
CREATE INDEX PersonIndex
ON Person (LastName)
```

如果您希望以降序索引某个列中的值，您可以在列名称之后添加保留字 *DESC*：

```
CREATE INDEX PersonIndex
ON Person (LastName DESC)
```

假如您希望索引不止一个列，您可以在括号中列出这些列的名称，用逗号隔开：

```
CREATE INDEX PersonIndex
ON Person (LastName, FirstName)
```

SQL 撤销索引、表以及数据库

通过使用 **DROP** 语句，可以轻松地删除索引、表和数据库。

SQL DROP INDEX 语句

我们可以使用 DROP INDEX 命令删除表格中的索引。

用于 **Microsoft SQLJet (**以及 **Microsoft Access)** 的语法**:**

```
DROP INDEX index_name ON table_name
```

用于 **MS SQL Server** 的语法**:**

```
DROP INDEX table_name.index_name
```

用于 **IBM DB2** 和 **Oracle** 语法**:**

```
DROP INDEX index_name
```

用于 **MySQL** 的语法**:**

```
ALTER TABLE table_name DROP INDEX index_name
```

SQL DROP TABLE 语句

DROP TABLE 语句用于删除表（表的结构、属性以及索引也会被删除）：

```
DROP TABLE 表名称
```

SQL DROP DATABASE 语句

DROP DATABASE 语句用于删除数据库：

```
DROP DATABASE 数据库名称
```

SQL TRUNCATE TABLE 语句

如果我们仅仅需要除去表内的数据，但并不删除表本身，那么我们该如何做呢？

请使用 TRUNCATE TABLE 命令（仅仅删除表格中的数据）：

```
TRUNCATE TABLE 表名称
```

SQL ALTER TABLE 语句

ALTER TABLE 语句

ALTER TABLE 语句用于在已有的表中添加、修改或删除列。

SQL ALTER TABLE 语法

如需在表中添加列，请使用下列语法:

```
ALTER TABLE table_name
ADD column_name datatype
```

要删除表中的列，请使用下列语法：

```
ALTER TABLE table_name
DROP COLUMN column_name
```

注释：某些数据库系统不允许这种在数据库表中删除列的方式 (DROP COLUMN column_name)。

要改变表中列的数据类型，请使用下列语法：

```
ALTER TABLE table_name
ALTER COLUMN column_name datatype
```

原始的表 (用在例子中的)：

Persons 表:

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

SQL ALTER TABLE 实例

现在，我们希望在表 "Persons" 中添加一个名为 "Birthday" 的新列。

我们使用下列 SQL 语句：

```
ALTER TABLE Persons
ADD Birthday date
```

请注意，新列 "Birthday" 的类型是 date，可以存放日期。数据类型规定列中可以存放的数据的类型。

新的 "Persons" 表类似这样：

Id	LastName	FirstName	Address	City	Birthday
1	Adams	John	Oxford Street	London	
2	Bush	George	Fifth Avenue	New York	
3	Carter	Thomas	Changan Street	Beijing	

改变数据类型实例

现在我们希望改变 "Persons" 表中 "Birthday" 列的数据类型。

我们使用下列 SQL 语句：

```
ALTER TABLE Persons
ALTER COLUMN Birthday year
```

请注意，"Birthday" 列的数据类型是 year，可以存放 2 位或 4 位格式的年份。

DROP COLUMN 实例

接下来，我们删除 "Person" 表中的 "Birthday" 列：

```
ALTER TABLE Person
DROP COLUMN Birthday
```

Persons 表会成为这样:

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

SQL AUTO INCREMENT 字段

Auto-increment 会在新记录插入表中时生成一个唯一的数字。

AUTO INCREMENT 字段

我们通常希望在每次插入新记录时，自动地创建主键字段的值。

我们可以在表中创建一个 auto-increment 字段。

用于 **MySQL** 的语法

下列 SQL 语句把 "Persons" 表中的 "P_Id" 列定义为 auto-increment 主键：

```
CREATE TABLE Persons
(
P_Id int NOT NULL AUTO_INCREMENT,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255),
PRIMARY KEY (P_Id)
)
```

MySQL 使用 AUTO_INCREMENT 关键字来执行 auto-increment 任务。

默认地，AUTO_INCREMENT 的开始值是 1，每条新记录递增 1。

要让 AUTO_INCREMENT 序列以其他的值起始，请使用下列 SQL 语法：

```
ALTER TABLE Persons AUTO_INCREMENT=100
```

要在 "Persons" 表中插入新记录，我们不必为 "P_Id" 列规定值（会自动添加一个唯一的值）：

```
INSERT INTO Persons (FirstName,LastName)
VALUES ('Bill','Gates')
```

上面的 SQL 语句会在 "Persons" 表中插入一条新记录。"P_Id" 会被赋予一个唯一的值。"FirstName" 会被设置为 "Bill"，"LastName" 列会被设置为 "Gates"。

用于 **SQL Server** 的语法

下列 SQL 语句把 "Persons" 表中的 "P_Id" 列定义为 auto-increment 主键：

```
CREATE TABLE Persons
(
P_Id int PRIMARY KEY IDENTITY,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255)
)
```

MS SQL 使用 IDENTITY 关键字来执行 auto-increment 任务。

默认地，IDENTITY 的开始值是 1，每条新记录递增 1。

要规定 "P_Id" 列以 20 起始且递增 10，请把 identity 改为 IDENTITY(20,10)

要在 "Persons" 表中插入新记录，我们不必为 "P_Id" 列规定值（会自动添加一个唯一的值）：

```
INSERT INTO Persons (FirstName,LastName)
VALUES ('Bill','Gates')
```

上面的 SQL 语句会在 "Persons" 表中插入一条新记录。"P_Id" 会被赋予一个唯一的值。"FirstName" 会被设置为 "Bill"，"LastName" 列会被设置为 "Gates"。

用于 **Access** 的语法

下列 SQL 语句把 "Persons" 表中的 "P_Id" 列定义为 auto-increment 主键：

```
CREATE TABLE Persons
(
P_Id int PRIMARY KEY AUTOINCREMENT,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255)
)
```

MS Access 使用 AUTOINCREMENT 关键字来执行 auto-increment 任务。

默认地，AUTOINCREMENT 的开始值是 1，每条新记录递增 1。

要规定 "P_Id" 列以 20 起始且递增 10，请把 autoincrement 改为 AUTOINCREMENT(20,10)

要在 "Persons" 表中插入新记录，我们不必为 "P_Id" 列规定值（会自动添加一个唯一的值）：

```
INSERT INTO Persons (FirstName,LastName)
VALUES ('Bill','Gates')
```

上面的 SQL 语句会在 "Persons" 表中插入一条新记录。"P_Id" 会被赋予一个唯一的值。"FirstName" 会被设置为 "Bill"，"LastName" 列会被设置为 "Gates"。

用于 Oracle 的语法

在 Oracle 中，代码稍微复杂一点。

您必须通过 sequence 对创建 auto-increment 字段（该对象生成数字序列）。

请使用下面的 CREATE SEQUENCE 语法：

```
CREATE SEQUENCE seq_person
MINVALUE 1
START WITH 1
INCREMENT BY 1
CACHE 10
```

上面的代码创建名为 seq_person 的序列对象，它以 1 起始且以 1 递增。该对象缓存 10 个值以提高性能。CACHE 选项规定了为了提高访问速度要存储多少个序列值。

要在 "Persons" 表中插入新记录，我们必须使用 nextval 函数（该函数从 seq_person 序列中取回下一个值）：

```
INSERT INTO Persons (P_Id,FirstName,LastName)
VALUES (seq_person.nextval,'Lars','Monsen')
```

上面的 SQL 语句会在 "Persons" 表中插入一条新记录。"P_Id" 的赋值是来自 seq_person 序列的下一个数字。"FirstName" 会被设置为 "Bill"，"LastName" 列会被设置为 "Gates"。

SQL VIEW（视图）

视图是可视化的表。

本章讲解如何创建、更新和删除视图。

SQL CREATE VIEW 语句

什么是视图？

在 SQL 中，视图是基于 SQL 语句的结果集的可视化的表。

视图包含行和列，就像一个真实的表。视图中的字段就是来自一个或多个数据库中的真实的表中的字段。我们可以向视图添加 SQL 函数、WHERE 以及 JOIN 语句，我们也可以提交数据，就像这些来自于某个单一的表。

注释：数据库的设计和结构不会受到视图中的函数、where 或 join 语句的影响。

SQL CREATE VIEW 语法

```
CREATE VIEW view_name AS
SELECT column_name(s)
FROM table_name
WHERE condition
```

注释：视图总是显示最近的数据。每当用户查询视图时，数据库引擎通过使用 SQL 语句来重建数据。

SQL CREATE VIEW 实例

可以从某个查询内部、某个存储过程内部，或者从另一个视图内部来使用视图。通过向视图添加函数、join 等等，我们可以向用户精确地提交我们希望提交的数据。

样本数据库 Northwind 拥有一些被默认安装的视图。视图 "Current Product List" 会从 Products 表列出所有正在使用的产品。这个视图使用下列 SQL 创建：

```
CREATE VIEW [Current Product List] AS
SELECT ProductID,ProductName
FROM Products
WHERE Discontinued=No
```

我们可以查询上面这个视图：

```
SELECT * FROM [Current Product List]
```

Northwind 样本数据库的另一个视图会选取 Products 表中所有单位价格高于平均单位价格的产品：

```
CREATE VIEW [Products Above Average Price] AS
SELECT ProductName,UnitPrice
FROM Products
WHERE UnitPrice>(SELECT AVG(UnitPrice) FROM Products)
```

我们可以像这样查询上面这个视图：

```
SELECT * FROM [Products Above Average Price]
```

另一个来自 Northwind 数据库的视图实例会计算在 1997 年每个种类的销售总数。请注意，这个视图会从另一个名为 "Product Sales for 1997" 的视图那里选取数据：

```
CREATE VIEW [Category Sales For 1997] AS
SELECT DISTINCT CategoryName,Sum(ProductSales) AS CategorySales
FROM [Product Sales for 1997]
GROUP BY CategoryName
```

我们可以像这样查询上面这个视图：

```
SELECT * FROM [Category Sales For 1997]
```

我们也可以向查询添加条件。现在，我们仅仅需要查看 "Beverages" 类的全部销量：

```
SELECT * FROM [Category Sales For 1997]
WHERE CategoryName='Beverages'
```

SQL 更新视图

您可以使用下面的语法来更新视图：

```
SQL CREATE OR REPLACE VIEW Syntax
CREATE OR REPLACE VIEW view_name AS
SELECT column_name(s)
FROM table_name
WHERE condition
```

现在，我们希望向 "Current Product List" 视图添加 "Category" 列。我们将通过下列 SQL 更新视图：

```
CREATE VIEW [Current Product List] AS
SELECT ProductID,ProductName,Category
FROM Products
WHERE Discontinued=No
```

SQL 撤销视图

您可以通过 DROP VIEW 命令来删除视图。

```
SQL DROP VIEW Syntax
DROP VIEW view_name
```

SQL函数

SQL Date 函数

SQL 日期

当我们处理日期时，最难的任务恐怕是确保所插入的日期的格式，与数据库中日期列的格式相匹配。

只要数据包含的只是日期部分，运行查询就不会出问题。但是，如果涉及时间，情况就有点复杂了。

在讨论日期查询的复杂性之前，我们先来看看最重要的内建日期处理函数。

MySQL Date 函数

下面的表格列出了 MySQL 中最重要的内建日期函数：

函数	描述
NOW()	返回当前的日期和时间
CURDATE()	返回当前的日期
CURTIME()	返回当前的时间
DATE()	提取日期或日期/时间表达式的日期部分
EXTRACT()	返回日期/时间按的单独部分
DATE_ADD()	给日期添加指定的时间间隔
DATE_SUB()	从日期减去指定的时间间隔
DATEDIFF()	返回两个日期之间的天数
DATE_FORMAT()	用不同的格式显示日期/时间

SQL Server Date 函数

下面的表格列出了 SQL Server 中最重要的内建日期函数：

函数	描述
GETDATE()	返回当前日期和时间
DATEPART()	返回日期/时间的单独部分
DATEADD()	在日期中添加或减去指定的时间间隔
DATEDIFF()	返回两个日期之间的时间
CONVERT()	用不同的格式显示日期/时间

SQL Date 数据类型

MySQL 使用下列数据类型在数据库中存储日期或日期/时间值：

- DATE - 格式 YYYY-MM-DD
- DATETIME - 格式: YYYY-MM-DD HH:MM:SS
- TIMESTAMP - 格式: YYYY-MM-DD HH:MM:SS
- YEAR - 格式 YYYY 或 YY

SQL Server 使用下列数据类型在数据库中存储日期或日期/时间值：

- DATE - 格式 YYYY-MM-DD
- DATETIME - 格式: YYYY-MM-DD HH:MM:SS
- SMALLDATETIME - 格式: YYYY-MM-DD HH:MM:SS
- TIMESTAMP - 格式: 唯一的数字

SQL 日期处理

如果不涉及时间部分，那么我们可以轻松地比较两个日期！

假设我们有下面这个 "Orders" 表：

OrderId	ProductName	OrderDate
1	computer	2008-12-26
2	printer	2008-12-26
3	electrograph	2008-11-12
4	telephone	2008-10-19

现在，我们希望从上表中选取 OrderDate 为 "2008-12-26" 的记录。

我们使用如下 SELECT 语句：

```
SELECT * FROM Orders WHERE OrderDate='2008-12-26'
```

结果集：

OrderId	ProductName	OrderDate
1	computer	2008-12-26
3	electrograph	2008-12-26

现在假设 "Orders" 类似这样（请注意 "OrderDate" 列中的时间部分）：

OrderId	ProductName	OrderDate
1	computer	2008-12-26 16:23:55
2	printer	2008-12-26 10:45:26
3	electrograph	2008-11-12 14:12:08
4	telephone	2008-10-19 12:56:10

如果我们使用上面的 SELECT 语句：

```
SELECT * FROM Orders WHERE OrderDate='2008-12-26'
```

那么我们得不到结果。这是由于该查询不含有时间部分的日期。

提示：如果您希望使查询简单且更易维护，那么请不要在日期中使用时间部分！

SQL NULL 值

NULL 值是遗漏的未知数据。

默认地，表的列可以存放 **NULL** 值。

本章讲解 **IS NULL** 和 **IS NOT NULL** 操作符。

SQL NULL 值

如果表中的某个列是可选的，那么我们可以在不向该列添加值的情况下插入新记录或更新已有的记录。这意味着该字段将以 NULL 值保存。

NULL 值的处理方式与其他值不同。

NULL 用作未知的或不适用的值的占位符。

注释：无法比较 NULL 和 0；它们是不等价的。

SQL 的 NULL 值处理

请看下面的 "Persons" 表：

Id	LastName	FirstName	Address	City
1	Adams	John	London	
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Beijing	

假如 "Persons" 表中的 "Address" 列是可选的。这意味着如果在 "Address" 列插入一条不带值的记录，"Address" 列会使用 NULL 值保存。

那么我们如何测试 NULL 值呢？

无法使用比较运算符来测试 NULL 值，比如 =, <, 或者 <>。

我们必须使用 IS NULL 和 IS NOT NULL 操作符。

SQL IS NULL

我们如何仅仅选取在 "Address" 列中带有 NULL 值的记录呢？

我们必须使用 IS NULL 操作符：

```
SELECT LastName,FirstName,Address FROM Persons
WHERE Address IS NULL
```

结果集：

LastName	FirstName	Address
Adams	John	
Carter	Thomas	

提示：请始终使用 IS NULL 来查找 NULL 值。

SQL IS NOT NULL

我们如何选取在 "Address" 列中不带有 NULL 值的记录呢？

我们必须使用 IS NOT NULL 操作符：

```
SELECT LastName,FirstName,Address FROM Persons
WHERE Address IS NOT NULL
```

结果集：

LastName	FirstName	Address
Bush	George	Fifth Avenue

在下一节中，我们了解 ISNULL()、NVL()、IFNULL() 和 COALESCE() 函数。

SQL NULL 函数

SQL ISNULL()、NVL()、IFNULL() 和 COALESCE() 函数

请看下面的 "Products" 表：

P_Id	ProductName	UnitPrice	UnitsInStock	UnitsOnOrder
1	computer	699	25	15
2	printer	365	36	
3	telephone	280	159	57

假如 "UnitsOnOrder" 是可选的，而且可以包含 NULL 值。

我们使用如下 SELECT 语句：

```
SELECT ProductName,UnitPrice*(UnitsInStock+UnitsOnOrder)
FROM Products
```

在上面的例子中，如果有 "UnitsOnOrder" 值是 NULL，那么结果是 NULL。

微软的 ISNULL() 函数用于规定如何处理 NULL 值。

NVL(), IFNULL() 和 COALESCE() 函数也可以达到相同的结果。

在这里，我们希望 NULL 值为 0。

下面，如果 "UnitsOnOrder" 是 NULL，则不利于计算，因此如果值是 NULL 则 ISNULL() 返回 0。

SQL Server / MS Access

```
SELECT ProductName,UnitPrice*(UnitsInStock+ISNULL(UnitsOnOrder,0))
FROM Products
```

Oracle

Oracle 没有 ISNULL() 函数。不过，我们可以使用 NVL() 函数达到相同的结果：

```
SELECT ProductName,UnitPrice*(UnitsInStock+NVL(UnitsOnOrder,0))
FROM Products
```

MySQL

MySQL 也拥有类似 ISNULL() 的函数。不过它的工作方式与微软的 ISNULL() 函数有点不同。

在 MySQL 中，我们可以使用 IFNULL() 函数，就像这样：

```
SELECT ProductName,UnitPrice*(UnitsInStock+IFNULL(UnitsOnOrder,0))
FROM Products
```

或者我们可以使用 COALESCE() 函数，就像这样：

```
SELECT ProductName,UnitPrice*(UnitsInStock+COALESCE(UnitsOnOrder,0
FROM Products
```

SQL 数据类型

Microsoft Access、MySQL 以及 **SQL Server** 所使用的数据类型和范围。

Microsoft Access 数据类型

数据类型	描述	存储
Text	用于文本或文本与数字的组合。最多 255 个字符。	
Memo	Memo 用于更大数量的文本。最多存储 65,536 个字符。 注释：无法对 memo 字段进行排序。不过它们是可搜索的。	
Byte	允许 0 到 255 的数字。	1 字节
Integer	允许介于 -32,768 到 32,767 之间的数字。	2 字节
Long	允许介于 -2,147,483,648 与 2,147,483,647 之间的全部数字	4 字节
Single	单精度浮点。处理大多数小数。	4 字节
Double	双精度浮点。处理大多数小数。	8 字节
Currency	用于货币。支持 15 位的元，外加 4 位小数。 提示：您可以选择使用哪个国家的货币。	8 字节
AutoNumber	AutoNumber 字段自动为每条记录分配数字，通常从 1 开始。	4 字节
Date/Time	用于日期和时间	8 字节
Yes/No	逻辑字段，可以显示为 Yes/No、True/False 或 On/Off。 在代码中，使用常量 True 和 False （等价于 1 和 0） 注释：Yes/No 字段中不允许 Null 值	1 比特
Ole Object	可以存储图片、音频、视频或其他 BLOBs (Binary Large OBjects)	最多 1GB
Hyperlink	包含指向其他文件的链接，包括网页。	
Lookup Wizard	允许你创建一个可从下列列表中进行选择的选项列表。	4 字节

MySQL 数据类型

在 MySQL 中，有三种主要的类型：文本、数字和日期/时间类型。

Text 类型：

数据类型	描述
CHAR(size)	保存固定长度的字符串（可包含字母、数字以及特殊字符）。在括号中指定字符串的长度。最多 255 个字符。
VARCHAR(size)	保存可变长度的字符串（可包含字母、数字以及特殊字符）。在括号中指定字符串的最大长度。最多 255 个字符。 注释：如果值的长度大于 255，则被转换为 TEXT 类型。
TINYTEXT	存放最大长度为 255 个字符的字符串。
TEXT	存放最大长度为 65,535 个字符的字符串。
BLOB	用于 BLOBs (Binary Large OBjects)。存放最多 65,535 字节的数据。
MEDIUMTEXT	存放最大长度为 16,777,215 个字符的字符串。
MEDIUMBLOB	用于 BLOBs (Binary Large OBjects)。存放最多 16,777,215 字节的数据。
LONGTEXT	存放最大长度为 4,294,967,295 个字符的字符串。
LONGBLOB	用于 BLOBs (Binary Large OBjects)。存放最多 4,294,967,295 字节的数据。
ENUM(x,y,z,etc.)	允许你输入可能值的列表。可以在 ENUM 列表中列出最大 65535 个值。如果列表中不存在插入的值，则插入空值。 注释：这些值是按照你输入的顺序存储的。 可以按照此格式输入可能的值：ENUM('X','Y','Z')
SET	与 ENUM 类似，SET 最多只能包含 64 个列表项，不过 SET 可存储一个以上的值。

Number 类型：

数据类型	描述
TINYINT(size)	-128 到 127 常规。0 到 255 无符号*。在括号中规定最大位数。
SMALLINT(size)	-32768 到 32767 常规。0 到 65535 无符号*。在括号中规定最大位数。
MEDIUMINT(size)	-8388608 到 8388607 普通。0 to 16777215 无符号*。在括号中规定最大位数。
INT(size)	-2147483648 到 2147483647 常规。0 到 4294967295 无符号*。在括号中规定最大位数。
BIGINT(size)	-9223372036854775808 到 9223372036854775807 常规。0 到 18446744073709551615 无符号*。在括号中规定最大位数。
FLOAT(size,d)	带有浮动小数点的小数字。在括号中规定最大位数。在 d 参数中规定小数点右侧的最大位数。
DOUBLE(size,d)	带有浮动小数点的大数字。在括号中规定最大位数。在 d 参数中规定小数点右侧的最大位数。
DECIMAL(size,d)	作为字符串存储的 DOUBLE 类型，允许固定的小数点。

- 这些整数类型拥有额外的选项 UNSIGNED。通常，整数可以是负数或正数。如果添加 UNSIGNED 属性，那么范围将从 0 开始，而不是某个负数。

Date 类型：

数据类型	描述
DATE()	日期。格式：YYYY-MM-DD 注释：支持的范围是从 '1000-01-01' 到 '9999-12-31'
DATETIME()	*日期和时间的组合。格式：YYYY-MM-DD HH:MM:SS 注释：支持的范围是从 '1000-01-01 00:00:00' 到 '9999-12-31 23:59:59'
TIMESTAMP()	*时间戳。TIMESTAMP 值使用 Unix 纪元('1970-01-01 00:00:00' UTC) 至今的描述来存储。格式：YYYY-MM-DD HH:MM:SS 注释：支持的范围是从 '1970-01-01 00:00:01' UTC 到 '2038-01-09 03:14:07' UTC
TIME()	时间。格式：HH:MM:SS 注释：支持的范围是从 '-838:59:59' 到 '838:59:59'
YEAR()	2 位或 4 位格式的年。 注释：4 位格式所允许的值：1901 到 2155。2 位格式所允许的值：70 到 69，表示从 1970 到 2069。

- 即便 DATETIME 和 TIMESTAMP 返回相同的格式，它们的工作方式很不同。在 INSERT 或 UPDATE 查询中，TIMESTAMP 自动把自身设置为当前的日期和时间。TIMESTAMP 也接受不同的格式，比如 YYYYMMDDHHMMSS、YYMMDDHHMMSS、YYYYMMDD 或 YYMMDD。

SQL Server 数据类型

Character 字符串：

数据类型	描述	存储
char(n)	固定长度的字符串。最多 8,000 个字符。	n
varchar(n)	可变长度的字符串。最多 8,000 个字符。	
varchar(max)	可变长度的字符串。最多 1,073,741,824 个字符。	
text	可变长度的字符串。最多 2GB 字符数据。	

Unicode 字符串：

数据类型	描述	存储
nchar(n)	固定长度的 Unicode 数据。最多 4,000 个字符。	
nvarchar(n)	可变长度的 Unicode 数据。最多 4,000 个字符。	
nvarchar(max)	可变长度的 Unicode 数据。最多 536,870,912 个字符。	
ntext	可变长度的 Unicode 数据。最多 2GB 字符数据。	

Binary 类型：

数据类型	描述	存储
bit	允许 0、1 或 NULL	
binary(n)	固定长度的二进制数据。最多 8,000 字节。	
varbinary(n)	可变长度的二进制数据。最多 8,000 字节。	
varbinary(max)	可变长度的二进制数据。最多 2GB 字节。	
image	可变长度的二进制数据。最多 2GB。	

Number 类型：

数据类型	描述	存储
tinyint	允许从 0 到 255 的所有数字。	1 字节
smallint	允许从 -32,768 到 32,767 的所有数字。	2 字节
int	允许从 -2,147,483,648 到 2,147,483,647 的所有数字。	4 字节
bigint	允许介于 -9,223,372,036,854,775,808 和 9,223,372,036,854,775,807 之间的所有数字。	8 字节
decimal(p,s)	固定精度和比例的数字。允许从 -10^38 +1 到 10^38 -1 之间的数字。 p 参数指示可以存储的最大位数（小数点左侧和右侧）。p 必须是 1 到 38 之间的值。默认是 18。 s 参数指示小数点右侧存储的最大位数。s 必须是 0 到 p 之间的值。默认是 0。	5-17 字节
numeric(p,s)	固定精度和比例的数字。允许从 -10^38 +1 到 10^38 -1 之间的数字。 p 参数指示可以存储的最大位数（小数点左侧和右侧）。p 必须是 1 到 38 之间的值。默认是 18。 s 参数指示小数点右侧存储的最大位数。s 必须是 0 到 p 之间的值。默认是 0。	5-17 字节
smallmoney	介于 -214,748.3648 和 214,748.3647 之间的货币数据。	4 字节
money	介于 -922,337,203,685,477.5808 和 922,337,203,685,477.5807 之间的货币数据。	8 字节
float(n)	从 -1.79E + 308 到 1.79E + 308 的浮动精度数字数据。 参数 n 指示该字段保存 4 字节还是 8 字节。float(24) 保存 4 字节，而 float(53) 保存 8 字节。n 的默认值是 53。	4 或 8 字节
real	从 -3.40E + 38 到 3.40E + 38 的浮动精度数字数据。	4 字节

Date 类型：

数据类型	描述	存储
datetime	从 1753 年 1 月 1 日 到 9999 年 12 月 31 日，精度为 3.33 毫秒。	8 bytes
datetime2	从 1753 年 1 月 1 日 到 9999 年 12 月 31 日，精度为 100 纳秒。	6-8 bytes
smalldatetime	从 1900 年 1 月 1 日 到 2079 年 6 月 6 日，精度为 1 分钟。	4 bytes
date	仅存储日期。从 0001 年 1 月 1 日 到 9999 年 12 月 31 日。	3 bytes
time	仅存储时间。精度为 100 纳秒。	3-5 bytes
datetimeoffset	与 datetime2 相同，外加时区偏移。	8-10 bytes
timestamp	存储唯一的数字，每当创建或修改某行时，该数字会更新。timestamp 基于内部时钟，不对应真实时间。每个表只能有一个 timestamp 变量。	

其他数据类型：

数据类型	描述
sql_variant	存储最多 8,000 字节不同数据类型的数据，除了 text、ntext 以及 timestamp。
uniqueidentifier	存储全局标识符 (GUID)。
xml	存储 XML 格式化数据。最多 2GB。
cursor	存储对用于数据库操作的指针的引用。
table	存储结果集，供稍后处理。

SQL 服务器 - RDBMS

现代的 **SQL** 服务器构建在 **RDBMS** 之上。

DBMS - 数据库管理系统（Database Management System）

数据库管理系统是一种可以访问数据库中数据的计算机程序。

DBMS 使我们有能力在数据库中提取、修改或者存贮信息。

不同的 DBMS 提供不同的函数供查询、提交以及修改数据。

RDBMS - 关系数据库管理系统（Relational Database Management System）

关系数据库管理系统 (RDBMS) 也是一种数据库管理系统，其数据库是根据数据间的关系来组织和访问数据的。

20 世纪 70 年代初，IBM 公司发明了 RDBMS。

RDBMS 是 SQL 的基础，也是所有现代数据库系统诸如 Oracle、SQL Server、IBM DB2、Sybase、MySQL 以及 Microsoft Access 的基础。

SQL 函数

SQL 拥有很多可用于计数和计算的内建函数。

函数的语法

内建 SQL 函数的语法是：

```
SELECT function(列) FROM 表
```

函数的类型

在 SQL 中，基本的函数类型和种类有若干种。函数的基本类型是：

- Aggregate 函数
- Scalar 函数

合计函数（**Aggregate functions**）

Aggregate 函数的操作面向一系列的值，并返回一个单一的值。

注释：如果在 SELECT 语句的项目列表中的众多其它表达式中使用 SELECT 语句，则这个 SELECT 必须使用 GROUP BY 语句！

"Persons" table (在大部分的例子中使用过)

Name	Age
Adams, John	38
Bush, George	33
Carter, Thomas	28

MS Access 中的合计函数

函数	描述
AVG(column)	返回某列的平均值
COUNT(column)	返回某列的行数（不包括 NULL 值）
COUNT(*)	返回被选行数
FIRST(column)	返回在指定的域中第一个记录的值
LAST(column)	返回在指定的域中最后一个记录的值
MAX(column)	返回某列的最高值
MIN(column)	返回某列的最低值
STDEV(column)	
STDEVP(column)	
SUM(column)	返回某列的总和
VAR(column)	
VARP(column)	

在 SQL Server 中的合计函数

函数	描述
AVG(column)	返回某列的平均值
BINARY_CHECKSUM	
CHECKSUM	
CHECKSUM_AGG	
COUNT(column)	返回某列的行数（不包括NULL值）
COUNT(*)	返回被选行数
COUNT(DISTINCT column)	返回相异结果的数目
FIRST(column)	返回在指定的域中第一个记录的值（SQLServer2000 不支持）
LAST(column)	返回在指定的域中最后一个记录的值（SQLServer2000 不支持）
MAX(column)	返回某列的最高值
MIN(column)	返回某列的最低值
STDEV(column)	
STDEVP(column)	
SUM(column)	返回某列的总和
VAR(column)	
VARP(column)	

Scalar 函数 Scalar 函数的操作面向某个单一的值，并返回基于输入值的一个单一的值。 ### MS Access 中的 Scalar 函数

函数	描述
UCASE(c)	将某个域转换为大写
LCASE(c)	将某个域转换为小写
MID(c,start[,end])	从某个文本域提取字符
LEN(c)	返回某个文本域的长度
INSTR(c,char)	返回在某个文本域中指定字符的数值位置
LEFT(c,number_of_char)	返回某个被请求的文本域的左侧部分
RIGHT(c,number_of_char)	返回某个被请求的文本域的右侧部分
ROUND(c,decimals)	对某个数值域进行指定小数位数的四舍五入
MOD(x,y)	返回除法操作的余数
NOW()	返回当前的系统日期
FORMAT(c,format)	改变某个域的显示方式
DATEDIFF(d,date1,date2)	用于执行日期计算

SQL AVG 函数

定义和用法

AVG 函数返回数值列的平均值。NULL 值不包括在计算中。

SQL AVG() 语法

```
SELECT AVG(column_name) FROM table_name
```

SQL AVG() 实例

我们拥有下面这个 "Orders" 表：

O_Id	OrderDate	OrderPrice	Customer
1	2008/12/29	1000	Bush
2	2008/11/23	1600	Carter
3	2008/10/05	700	Bush
4	2008/09/28	300	Bush
5	2008/08/06	2000	Adams
6	2008/07/21	100	Carter

例子 1

现在，我们希望计算 "OrderPrice" 字段的平均值。

我们使用如下 SQL 语句：

```
SELECT AVG(OrderPrice) AS OrderAverage FROM Orders
```

结果集类似这样：

OrderAverage
950

例子 2

现在，我们希望找到 OrderPrice 值高于 OrderPrice 平均值的客户。

我们使用如下 SQL 语句：

```
SELECT Customer FROM Orders
WHERE OrderPrice>(SELECT AVG(OrderPrice) FROM Orders)
```

结果集类似这样：

Customer
Bush
Carter
Adams

SQL COUNT() 函数

COUNT() 函数返回匹配指定条件的行数。

SQL COUNT() 语法

SQL COUNT(column_name) 语法

COUNT(column_name) 函数返回指定列的值的数目（NULL 不计入）：

```
SELECT COUNT(column_name) FROM table_name
```

SQL COUNT(*) 语法

COUNT(*) 函数返回表中的记录数：

```
SELECT COUNT(*) FROM table_name
```

SQL COUNT(DISTINCT column_name) 语法

COUNT(DISTINCT column_name) 函数返回指定列的不同值的数目：

```
SELECT COUNT(DISTINCT column_name) FROM table_name
```

注释：COUNT(DISTINCT) 适用于 ORACLE 和 Microsoft SQL Server，但是无法用于 Microsoft Access。

SQL COUNT(column_name) 实例

我们拥有下列 "Orders" 表：

O_Id	OrderDate	OrderPrice	Customer
1	2008/12/29	1000	Bush
2	2008/11/23	1600	Carter
3	2008/10/05	700	Bush
4	2008/09/28	300	Bush
5	2008/08/06	2000	Adams
6	2008/07/21	100	Carter

现在，我们希望计算客户 "Carter" 的订单数。

我们使用如下 SQL 语句：

```
SELECT COUNT(Customer) AS CustomerNilsen FROM Orders
WHERE Customer='Carter'
```

以上 SQL 语句的结果是 2，因为客户 Carter 共有 2 个订单：

CustomerNilsen
2

SQL COUNT(*) 实例

如果我们省略 WHERE 子句，比如这样：

```
SELECT COUNT(*) AS NumberOfOrders FROM Orders
```

结果集类似这样：

NumberOfOrders
6

这是表中的总行数。

SQL COUNT(DISTINCT column_name) 实例

现在，我们希望计算 "Orders" 表中不同客户的数目。

我们使用如下 SQL 语句：

```
SELECT COUNT(DISTINCT Customer) AS NumberOfCustomers FROM Orders
```

结果集类似这样：

NumberOfCustomers
3

这是 "Orders" 表中不同客户（Bush, Carter 和 Adams）的数目。

SQL FIRST() 函数

FIRST() 函数

FIRST() 函数返回指定的字段中第一个记录的值。

提示：可使用 ORDER BY 语句对记录进行排序。

SQL FIRST() 语法

```
SELECT FIRST(column_name) FROM table_name
```

SQL FIRST() 实例

我们拥有下面这个 "Orders" 表：

O_Id	OrderDate	OrderPrice	Customer
1	2008/12/29	1000	Bush
2	2008/11/23	1600	Carter
3	2008/10/05	700	Bush
4	2008/09/28	300	Bush
5	2008/08/06	2000	Adams
6	2008/07/21	100	Carter

现在，我们希望查找 "OrderPrice" 列的第一个值。

我们使用如下 SQL 语句：

```
SELECT FIRST(OrderPrice) AS FirstOrderPrice FROM Orders
```

结果集类似这样：

FirstOrderPrice
1000

SQL LAST() 函数

LAST() 函数

LAST() 函数返回指定的字段中最后一个记录的值。

提示：可使用 ORDER BY 语句对记录进行排序。

SQL LAST() 语法

```
SELECT LAST(column_name) FROM table_name
```

SQL LAST() 实例

我们拥有下面这个 "Orders" 表：

O_Id	OrderDate	OrderPrice	Customer
1	2008/12/29	1000	Bush
2	2008/11/23	1600	Carter
3	2008/10/05	700	Bush
4	2008/09/28	300	Bush
5	2008/08/06	2000	Adams
6	2008/07/21	100	Carter

现在，我们希望查找 "OrderPrice" 列的最后一个值。

我们使用如下 SQL 语句：

```
SELECT LAST(OrderPrice) AS LastOrderPrice FROM Orders
```

结果集类似这样：

LastOrderPrice
100

SQL MAX() 函数

MAX() 函数

MAX 函数返回一列中的最大值。NULL 值不包括在计算中。

SQL MAX() 语法

```
SELECT MAX(column_name) FROM table_name
```

注释：MIN 和 MAX 也可用于文本列，以获得按字母顺序排列的最高或最低值。

SQL MAX() 实例

我们拥有下面这个 "Orders" 表：

O_Id	OrderDate	OrderPrice	Customer
1	2008/12/29	1000	Bush
2	2008/11/23	1600	Carter
3	2008/10/05	700	Bush
4	2008/09/28	300	Bush
5	2008/08/06	2000	Adams
6	2008/07/21	100	Carter

现在，我们希望查找 "OrderPrice" 列的最大值。

我们使用如下 SQL 语句：

```
SELECT MAX(OrderPrice) AS LargestOrderPrice FROM Orders
```

结果集类似这样：

LargestOrderPrice
2000

SQL MIN() 函数

MIN() 函数

MIN 函数返回一列中的最小值。NULL 值不包括在计算中。

SQL MIN() 语法

```
SELECT MIN(column_name) FROM table_name
```

注释：MIN 和 MAX 也可用于文本列，以获得按字母顺序排列的最高或最低值。

SQL MIN() 实例

我们拥有下面这个 "Orders" 表：

O_Id	OrderDate	OrderPrice	Customer
1	2008/12/29	1000	Bush
2	2008/11/23	1600	Carter
3	2008/10/05	700	Bush
4	2008/09/28	300	Bush
5	2008/08/06	2000	Adams
6	2008/07/21	100	Carter

现在，我们希望查找 "OrderPrice" 列的最小值。

我们使用如下 SQL 语句：

```
SELECT MIN(OrderPrice) AS SmallestOrderPrice FROM Orders
```

结果集类似这样：

SmallestOrderPrice
100

SQL SUM() 函数

SUM() 函数

SUM 函数返回数值列的总数（总额）。

SQL SUM() 语法

```
SELECT SUM(column_name) FROM table_name
```

SQL SUM() 实例

我们拥有下面这个 "Orders" 表：

O_Id	OrderDate	OrderPrice	Customer
1	2008/12/29	1000	Bush
2	2008/11/23	1600	Carter
3	2008/10/05	700	Bush
4	2008/09/28	300	Bush
5	2008/08/06	2000	Adams
6	2008/07/21	100	Carter

现在，我们希望查找 "OrderPrice" 字段的总数。

我们使用如下 SQL 语句：

```
SELECT SUM(OrderPrice) AS OrderTotal FROM Orders
```

结果集类似这样：

OrderTotal
5700

SQL GROUP BY 语句

合计函数 (比如 **SUM)** 常常需要添加 **GROUP BY** 语句。

GROUP BY 语句

GROUP BY 语句用于结合合计函数，根据一个或多个列对结果集进行分组。

SQL GROUP BY 语法

```
SELECT column_name, aggregate_function(column_name)
FROM table_name
WHERE column_name operator value
GROUP BY column_name
```

SQL GROUP BY 实例

我们拥有下面这个 "Orders" 表：

O_Id	OrderDate	OrderPrice	Customer
1	2008/12/29	1000	Bush
2	2008/11/23	1600	Carter
3	2008/10/05	700	Bush
4	2008/09/28	300	Bush
5	2008/08/06	2000	Adams
6	2008/07/21	100	Carter

现在，我们希望查找每个客户的总金额（总订单）。

我们想要使用 GROUP BY 语句对客户进行组合。

我们使用下列 SQL 语句：

```
SELECT Customer,SUM(OrderPrice) FROM Orders
GROUP BY Customer
```

结果集类似这样：

Customer	SUM(OrderPrice)
Bush	2000
Carter	1700
Adams	2000

很棒吧，对不对？

让我们看一下如果省略 GROUP BY 会出现什么情况：

```
SELECT Customer,SUM(OrderPrice) FROM Orders
```

结果集类似这样：

Customer	SUM(OrderPrice)
Bush	5700
Carter	5700
Bush	5700
Bush	5700
Adams	5700
Carter	5700

上面的结果集不是我们需要的。

那么为什么不能使用上面这条 SELECT 语句呢？解释如下：上面的 SELECT 语句指定了两列（Customer 和 SUM(OrderPrice)）。"SUM(OrderPrice)" 返回一个单独的值（"OrderPrice" 列的总计），而 "Customer" 返回 6 个值（每个值对应 "Orders" 表中的每一行）。因此，我们得不到正确的结果。不过，您已经看到了，GROUP BY 语句解决了这个问题。

GROUP BY 一个以上的列

我们也可以对一个以上的列应用 GROUP BY 语句，就像这样：

```
SELECT Customer,OrderDate,SUM(OrderPrice) FROM Orders
GROUP BY Customer,OrderDate
```

SQL HAVING 子句

HAVING 子句

在 SQL 中增加 HAVING 子句原因是，WHERE 关键字无法与合计函数一起使用。

SQL HAVING 语法

```
SELECT column_name, aggregate_function(column_name)
FROM table_name
WHERE column_name operator value
GROUP BY column_name
HAVING aggregate_function(column_name) operator value
```

SQL HAVING 实例

我们拥有下面这个 "Orders" 表：

O_Id	OrderDate	OrderPrice	Customer
1	2008/12/29	1000	Bush
2	2008/11/23	1600	Carter
3	2008/10/05	700	Bush
4	2008/09/28	300	Bush
5	2008/08/06	2000	Adams
6	2008/07/21	100	Carter

现在，我们希望查找订单总金额少于 2000 的客户。

我们使用如下 SQL 语句：

```
SELECT Customer,SUM(OrderPrice) FROM Orders
GROUP BY Customer
HAVING SUM(OrderPrice)<2000
```

结果集类似：

Customer	SUM(OrderPrice)
Carter	1700

现在我们希望查找客户 "Bush" 或 "Adams" 拥有超过 1500 的订单总金额。

我们在 SQL 语句中增加了一个普通的 WHERE 子句：

```
SELECT Customer,SUM(OrderPrice) FROM Orders
WHERE Customer='Bush' OR Customer='Adams'
GROUP BY Customer
HAVING SUM(OrderPrice)>1500
```

结果集：

Customer	SUM(OrderPrice)
Bush	2000
Adams	2000

SQL UCASE() 函数

UCASE() 函数

UCASE 函数把字段的值转换为大写。

SQL UCASE() 语法

```
SELECT UCASE(column_name) FROM table_name
```

SQL UCASE() 实例

我们拥有下面这个 "Persons" 表：

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

现在，我们希望选取 "LastName" 和 "FirstName" 列的内容，然后把 "LastName" 列转换为大写。

我们使用如下 SQL 语句：

```
SELECT UCASE(LastName) as LastName,FirstName FROM Persons
```

结果集类似这样：

LastName	FirstName
ADAMS	John
BUSH	George
CARTER	Thomas

SQL LCASE() 函数

LCASE() 函数

LCASE 函数把字段的值转换为小写。

SQL LCASE() 语法

```
SELECT LCASE(column_name) FROM table_name
```

SQL LCASE() 实例

我们拥有下面这个 "Persons" 表：

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

现在，我们希望选取 "LastName" 和 "FirstName" 列的内容，然后把 "LastName" 列转换为小写。

我们使用如下 SQL 语句：

```
SELECT LCASE(LastName) as LastName,FirstName FROM Persons
```

结果集类似这样：

LastName	FirstName
adams	John
bush	George
carter	Thomas

SQL MID() 函数

MID() 函数

MID 函数用于从文本字段中提取字符。

SQL MID() 语法

```
SELECT MID(column_name,start[,length]) FROM table_name
```

参数	描述
column_name	必需。要提取字符的字段。
start	必需。规定开始位置（起始值是 1）。
length	可选。要返回的字符数。如果省略，则 MID() 函数返回剩余文本。

SQL MID() 实例

我们拥有下面这个 "Persons" 表：

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

现在，我们希望从 "City" 列中提取前 3 个字符。

我们使用如下 SQL 语句：

```
SELECT MID(City,1,3) as SmallCity FROM Persons
```

结果集类似这样：

SmallCity
Lon
New
Bei

SQL LEN() 函数

LEN() 函数

LEN 函数返回文本字段中值的长度。

SQL LEN() 语法

```
SELECT LEN(column_name) FROM table_name
```

SQL LEN() 实例

我们拥有下面这个 "Persons" 表：

Id	LastName	FirstName	Address	City
1	Adams	John	Oxford Street	London
2	Bush	George	Fifth Avenue	New York
3	Carter	Thomas	Changan Street	Beijing

现在，我们希望取得 "City" 列中值的长度。

我们使用如下 SQL 语句：

```
SELECT LEN(City) as LengthOfCity FROM Persons
```

结果集类似这样：

LengthOfCity
6
8
7

SQL ROUND() 函数

ROUND() 函数

ROUND 函数用于把数值字段舍入为指定的小数位数。

SQL ROUND() 语法

```
SELECT ROUND(column_name,decimals) FROM table_name
```

参数	描述
column_name	必需。要舍入的字段。
decimals	必需。规定要返回的小数位数。

SQL ROUND() 实例

我们拥有下面这个 "Products" 表：

Prod_Id	ProductName	Unit	UnitPrice
1	gold	1000 g	32.35
2	silver	1000 g	11.56
3	copper	1000 g	6.85

现在，我们希望把名称和价格舍入为最接近的整数。

我们使用如下 SQL 语句：

```
SELECT ProductName, ROUND(UnitPrice,0) as UnitPrice FROM Products
```

结果集类似这样：

ProductName	UnitPrice
gold	32
silver	12
copper	7

SQL NOW() 函数

NOW() 函数

NOW 函数返回当前的日期和时间。

提示：如果您在使用 Sql Server 数据库，请使用 getdate() 函数来获得当前的日期时间。

SQL NOW() 语法

```
SELECT NOW() FROM table_name
```

SQL NOW() 实例

我们拥有下面这个 "Products" 表：

Prod_Id	ProductName	Unit	UnitPrice
1	gold	1000 g	32.35
2	silver	1000 g	11.56
3	copper	1000 g	6.85

现在，我们希望显示当天的日期所对应的名称和价格。

我们使用如下 SQL 语句：

```
SELECT ProductName, UnitPrice, Now() as PerDate FROM Products
```

结果集类似这样：

ProductName	UnitPrice	PerDate
gold	32.35	12/29/2008 11:36:05 AM
silver	11.56	12/29/2008 11:36:05 AM
copper	6.85	12/29/2008 11:36:05 AM

SQL FORMAT() 函数

FORMAT() 函数

FORMAT 函数用于对字段的显示进行格式化。

SQL FORMAT() 语法

```
SELECT FORMAT(column_name,format) FROM table_name
```

参数	描述
column_name	必需。要格式化的字段。
format	必需。规定格式。

SQL FORMAT() 实例

我们拥有下面这个 "Products" 表：

Prod_Id	ProductName	Unit	UnitPrice
1	gold	1000 g	32.35
2	silver	1000 g	11.56
3	copper	1000 g	6.85

现在，我们希望显示每天日期所对应的名称和价格（日期的显示格式是 "YYYY-MM-DD"）。

我们使用如下 SQL 语句：

```
SELECT ProductName, UnitPrice, FORMAT(Now(),'YYYY-MM-DD') as PerDat
FROM Products
```

结果集类似这样：

ProductName	UnitPrice	PerDate
gold	32.35	12/29/2008
silver	11.56	12/29/2008
copper	6.85	12/29/2008

SQL 快速参考

来自 **W3School** 的 **SQL** 快速参考。可以打印它，以备日常使用。

SQL 语句

语句	语法
AND / OR	SELECT column_name(s) FROM table_name WHERE condition AND|OR condition
ALTER TABLE (add column)	ALTER TABLE table_name ADD column_name datatype
ALTER TABLE (drop column)	ALTER TABLE table_name DROP COLUMN column_name
AS (alias for column)	SELECT column_name AS column_alias FROM table_name
AS (alias for table)	SELECT column_name FROM table_name AS table_alias
BETWEEN	SELECT column_name(s) FROM table_name WHERE column_name BETWEEN value1 AND value2
CREATE DATABASE	CREATE DATABASE database_name
CREATE INDEX	CREATE INDEX index_name ON table_name (column_name)
CREATE TABLE	CREATE TABLE table_name (column_name1 data_type, column_name2 data_type, ...)
CREATE UNIQUE INDEX	CREATE UNIQUE INDEX index_name ON table_name (column_name)
CREATE VIEW	CREATE VIEW view_name AS SELECT column_name(s) FROM table_name WHERE condition
DELETE FROM	DELETE FROM table*name (**Note:** Deletes the entire table!!) _or* DELETE FROM table_name WHERE condition
DROP DATABASE	DROP DATABASE database_name
DROP INDEX	DROP INDEX table_name.index_name
DROP TABLE	DROP TABLE table_name

GROUP BY	SELECT column_name1,SUM(column_name2) FROM table_name GROUP BY column_name1
HAVING	SELECT column_name1,SUM(column_name2) FROM table_name GROUP BY column_name1 HAVING SUM(column_name2) condition value
IN	SELECT column_name(s) FROM table_name WHERE column_name IN (value1,value2,..)
INSERT INTO	INSERT INTO table*name VALUES (value1, value2,....)* _*or* INSERT INTO table_name (column_name1, column_name2,...) VALUES (value1, value2,....)
LIKE	SELECT column_name(s) FROM table_name WHERE column_name LIKE pattern
ORDER BY	SELECT column_name(s) FROM table_name ORDER BY column_name [ASC|DESC]
SELECT	SELECT column_name(s) FROM table_name
SELECT *	SELECT * FROM table_name
SELECT DISTINCT	SELECT DISTINCT column_name(s) FROM table_name
SELECT INTO (used to create backup copies of tables)	SELECT * INTO new*table_name FROM original_table_name* _*or* SELECT column_name(s) INTO new_table_name FROM original_table_name
TRUNCATE TABLE (deletes only the data inside the table)	TRUNCATE TABLE table_name
UPDATE	UPDATE table_name SET column_name=new_value [, column_name=new_value] WHERE column_name=some_value
WHERE	SELECT column_name(s) FROM table_name WHERE condition

MySQL 教程

MySQL 教程



Mysql是最流行的关系型数据库管理系统，在WEB应用方面MySQL是最好的RDBMS(Relational Database Management System：关系数据库管理系统)应用软件之一。

在本教程中，会让大家快速掌握Mysql的基本知识，并轻松使用Mysql数据库。

什么是数据库？

数据库（Database）是按照数据结构来组织、存储和管理数据的仓库，

每个数据库都有一个或多个不同的API用于创建，访问，管理，搜索和复制所保存的数据。

我们也可以将数据存储在文件中，但是在文件中读写数据速度相对较慢。

所以，现在我们使用关系型数据库管理系统（RDBMS）来存储和管理的大数据量。所谓的关系型数据库，是建立在关系模型基础上的数据库，借助于集合代数等数学概念和方法来处理数据库中的数据。

RDBMS即关系数据库管理系统(Relational Database Management System)的特点：

- 1.数据以表格的形式出现
- 2.每行为各种记录名称
- 3.每列为记录名称所对应的数据域
- 4.许多的行和列组成一张表单
- 5.若干的表单组成database

RDBMS 术语

在我们开始学习MySQL 数据库前，让我们先了解下RDBMS的一些术语：

- **数据库:** 数据库是一些关联表的集合。.

- 数据表: 表是数据的矩阵。在一个数据库中的表看起来像一个简单的电子表格。
- 列: 一列(数据元素) 包含了相同的数据, 例如邮政编码的数据。
- 行：一行（=元组，或记录）是一组相关的数据，例如一条用户订阅的数据。
- 冗余：存储两倍数据，冗余可以使系统速度更快。
- 主键：主键是唯一的。一个数据表中只能包含一个主键。你可以使用主键来查询数据。
- 外键：外键用于关联两个表。
- 复合键：复合键（组合键）将多个列作为一个索引键，一般用于复合索引。
- 索引：使用索引可快速访问数据库表中的特定信息。索引是对数据库表中一列或多列的值进行排序的一种结构。类似于书籍的目录。
- 参照完整性: 参照的完整性要求关系中不允许引用不存在的实体。与实体完整性是关系模型必须满足的完整性约束条件，目的是保证数据的一致性。

Mysql数据库

MySQL是一个关系型数据库管理系统，由瑞典MySQL AB公司开发，目前属于Oracle公司。MySQL是一种关联数据库管理系统，关联数据库将数据保存在不同的表中，而不是将所有数据放在一个大仓库内，这样就增加了速度并提高了灵活性。

- Mysql是开源的，所以你不需要支付额外的费用。
- Mysql支持大型的数据库。可以处理拥有上千万条记录的大型数据库。
- MySQL使用标准的SQL数据语言形式。
- Mysql可以允许于多个系统上，并且支持多种语言。这些编程语言包括C、C++、Python、Java、Perl、PHP、Eiffel、Ruby和Tcl等。
- Mysql对PHP有很好的支持，PHP是目前最流行的Web开发语言。
- MySQL支持大型数据库，支持5000万条记录的数据仓库，32位系统表文件最大可支持4GB，64位系统支持最大的表文件为8TB。
- Mysql是可以定制的，采用了GPL协议，你可以修改源码来开发自己的Mysql系统。

在开始学习本教程前你应该了解？

在开始学习本教程前你应该了解PHP和HTML的基础知识，并能简单的应用。

本教程的很多例子都跟PHP语言有关，我们的实例基本上是采用PHP语言来演示。

如果你还不了解PHP，你可以通过本站的PHP教程来了解该语言。

MySQL 安装

所有平台的Mysql下载地址为： MySQL 下载. 挑选你需要的 *MySQL Community Server* 版本及对应的平台。

Linux/UNIX上安装Mysql

Linux平台上推荐使用RPM包来安装Mysql,MySQL AB提供了以下RPM包的下载地址：

- **MySQL** - MySQL服务器。你需要该选项，除非你只想连接运行在另一台机器上的MySQL服务器。
- **MySQL-client** - MySQL 客户端程序，用于连接并操作Mysql服务器。
- **MySQL-devel** - 库和包含文件，如果你想要编译其它MySQL客户端，例如Perl模块，则需要安装该RPM包。
- **MySQL-shared** - 该软件包包含某些语言和应用程序需要动态装载的共享库(libmysqlclient.so*)，使用MySQL。
- **MySQL-bench** - MySQL数据库服务器的基准和性能测试工具。

以下安装Mysql RMP的实例是在SuSE Linux系统上进行，当然该安装步骤也适合应用于其他支持RPM的Linux系统，如:Centos。

安装步骤如下：

使用root用户登陆你的Linux系统。

下载Mysql RPM包，下载地址为：MySQL 下载。

通过以下命令执行Mysql安装，rpm包为你下载的rpm包：

```
[root@host]# rpm -i MySQL-5.0.9-0.i386.rpm
```

以上安装mysql服务器的过程会创建mysql用户，并创建一个mysql配置文件my.cnf。

你可以在/usr/bin和/usr/sbin中找到所有与MySQL相关的二进制文件。所有数据表和数据库将在/var/lib/mysql目录中创建。

以下是一些mysql可选包的安装过程，你可以根据自己的需要来安装：

```
[root@host]# rpm -i MySQL-client-5.0.9-0.i386.rpm
[root@host]# rpm -i MySQL-devel-5.0.9-0.i386.rpm
[root@host]# rpm -i MySQL-shared-5.0.9-0.i386.rpm
[root@host]# rpm -i MySQL-bench-5.0.9-0.i386.rpm
```

Window上安装Mysql

Window上安装Mysql相对来说会较为简单，你只需要载 MySQL 下载中下载window版本的mysql安装包，并解压安装包。

双击 setup.exe 文件，接下来你只需要安装默认的配置点击"next"即可，默认情况下安装信息会在C:\mysql目录中。

接下来你可以通过"开始" =》在搜索框中输入 " cmd" 命令 =》 在命令提示符上切换到 C:\mysql\bin 目录，并输入一下命令：

```
mysqld.exe --console
```

如果安装成功以上命令将输出一些mysql启动及InnoDB信息。

验证Mysql安装

在成功安装Mysql后，一些基础表会表初始化，在服务器启动后，你可以通过简单的测试来验证Mysql是否工作正常。

使用 mysqladmin 工具来获取服务器状态：

使用 mysqladmin 命令俩检查服务器的版本,在linux上该二进制文件位于 /usr/bin on linux ，在window上该二进制文件位于C:\mysql\bin 。

```
[root@host]# mysqladmin --version
```

linux上该命令将输出以下结果，该结果基于你的系统信息：

```
mysqladmin  Ver 8.23 Distrib 5.0.9-0, for redhat-linux-gnu on i386
```

如果以上命令执行后未输入任何信息，说明你的Mysql未安装成功。

使用 MySQL Client(Mysql客户端) 执行简单的SQL命令

你可以在 MySQL Client(Mysql客户端) 使用 mysql 命令连接到Mysql服务器上，默认情况下Mysql服务器的密码为空，所以本实例不需要输入密码。

命令如下：

```
[root@host]# mysql
```

以上命令执行后会输出 mysql>提示符，这说明你已经成功连接到Mysql服务器上，你可以在 mysql> 提示符执行SQL命令：

```
mysql> SHOW DATABASES;
+----------+
| Database |
+----------+
| mysql    |
| test     |
+----------+
2 rows in set (0.13 sec)
```

Mysql安装后需要做的

Mysql安装成功后，默认的root用户密码为空，你可以使用以下命令来创建root用户的密码：

```
[root@host]# mysqladmin -u root password "new_password";
```

现在你可以通过以下命令来连接到Mysql服务器：

```
[root@host]# mysql -u root -p
Enter password:*******
```

注意：在输入密码时，密码是不会显示了，你正确输入即可。

Linux系统启动时启动 MySQL

如果你需要在Linux系统启动时启动 MySQL 服务器，你需要在 /etc/rc.local 文件中添加以下命令：

```
/etc/init.d/mysqld start
```

同样，你需要将 mysqld 二进制文件添加到 /etc/init.d/ 目录中。

MySQL 管理

启动及关闭 MySQL 服务器

首先，我们需要通过以下命令来检查MySQL服务器是否启动：

```
ps -ef | grep mysqld
```

如果MySql已经启动，以上命令将输出mysql进程列表， 如果mysql未启动，你可以使用以下命令来启动mysql服务器:

```
root@host# cd /usr/bin
./safe_mysqld &
```

如果你想关闭目前运行的 MySQL 服务器, 你可以执行以下命令:

```
root@host# cd /usr/bin
./mysqladmin -u root -p shutdown
Enter password: ******
```

MySQL 用户设置

如果你需要添加 MySQL 用户，你只需要在 mysql 数据库中的 user 表添加新用户即可。

以下为添加用户的的实例，用户名为guest，密码为guest123，并授权用户可进行 SELECT, INSERT 和 UPDATE操作权限：

```
root@host# mysql -u root -p
Enter password:*******
mysql> use mysql;
Database changed

mysql> INSERT INTO user
          (host, user, password,
           select_priv, insert_priv, update_priv)
           VALUES ('localhost', 'guest',
           PASSWORD('guest123'), 'Y', 'Y', 'Y');
Query OK, 1 row affected (0.20 sec)

mysql> FLUSH PRIVILEGES;
Query OK, 1 row affected (0.01 sec)

mysql> SELECT host, user, password FROM user WHERE user = 'guest';
+-----------+---------+------------------+
| host      | user    | password         |
+-----------+---------+------------------+
| localhost | guest | 6f8c114b58f2ce9e |
+-----------+---------+------------------+
1 row in set (0.00 sec)
```

在添加用户时，请注意使用MySQL提供的 PASSWORD() 函数来对密码进行加密。你可以在以上实例看到用户密码加密后为： 6f8c114b58f2ce9e.

注意：在注意需要执行 FLUSH PRIVILEGES 语句。 这个命令执行后会重新载入授权表。 如果你不使用该命令，你就无法使用新创建的用户来连接mysql服务器，除非你重启mysql服务器。

你可以在创建用户时，为用户指定权限，在对应的权限列中，在插入语句中设置为 'Y' 即可，用户权限列表如下：

- Select_priv
- Insert_priv
- Update_priv
- Delete_priv
- Create_priv
- Drop_priv
- Reload_priv
- Shutdown_priv
- Process_priv
- File_priv
- Grant_priv
- References_priv
- Index_priv
- Alter_priv

另外一种添加用户的方法为通过SQL的 GRANT 命令，你下命令会给指定数据库TUTORIALS添加用户 zara ，密码为 zara123 。

```
root@host# mysql -u root -p password;
Enter password:*******
mysql> use mysql;
Database changed

mysql> GRANT SELECT,INSERT,UPDATE,DELETE,CREATE,DROP
    -> ON TUTORIALS.*
    -> TO 'zara'@'localhost'
    -> IDENTIFIED BY 'zara123';
```

以上命令会在mysql数据库中的user表创建一条用户信息记录。

注意**:** MySQL 的SQL语句以分号 (;) 作为结束标识。

/etc/my.cnf 文件配置

一般情况下，你不需要修改该配置文件，该文件默认配置如下：

```
[mysqld]
datadir=/var/lib/mysql
socket=/var/lib/mysql/mysql.sock

[mysql.server]
user=mysql
basedir=/var/lib

[safe_mysqld]
err-log=/var/log/mysqld.log
pid-file=/var/run/mysqld/mysqld.pid
```

在配置文件中，你可以指定不同的错误日志文件存放的目录，一般你不需要改动这些配置。

管理MySQL的命令

以下列出了使用Mysql数据库过程中常用的命令：

- **USE** 数据库名 :选择要操作的Mysql数据库，使用该命令后所有Mysql命令都只针对该数据库。
- **SHOW DATABASES:** 列出 MySQL 数据库管理系统的数据库列表。
- **SHOW TABLES:** 显示指定数据库的所有表，使用该命令前需要使用 use 命令来选择要操作的数据库。
- **SHOW COLUMNS FROM** 数据表**:** 显示数据表的属性，属性类型，主键信息

，是否为 NULL，默认值等其他信息。

- **SHOW INDEX FROM** 数据表**:** 显示数据表的详细索引信息，包括PRIMARY KEY（主键）。
- **SHOW TABLE STATUS LIKE** 数据表**\G:** 该命令将输出Mysql数据库管理系统的性能及统计信息。

MySQL PHP 语法

MySQL 可应用于多种语言，包括 PERL, C, C++, JAVA 和 PHP。 在这些语言中，Mysql在PHP的web开发中是应用最广泛。

在本教程中我们大部分实例都采用了PHP语言。你过你想了解Mysql在PHP中的应用，可以访问我们的 PHP中使用Mysql介绍。

PHP提供了多种方式来访问和操作Mysql数据库记录。PHP Mysql函数格式如下：

```
mysql_function(value,value,...);
```

以上格式中 function部分描述了mysql函数的功能，如

```
mysqli_connect($connect);
mysqli_query($connect,"SQL statement");
mysql_fetch_array()
mysql_connect(),mysql_close()
```

以下实例展示了PHP调用mysql函数的语法：

```
<html>
<head>
<title>PHP with MySQL</title>
</head>
<body>
<?php
   $retval = mysql_function(value, [value,...]);
   if( !$retval )
   {
       die ( "Error: a related error message" );
   }
   // Otherwise MySQL  or PHP Statements
?>
</body>
</html>
```

从下一章开始，我们将学习到更多的MySQL功能函数。

MySQL 连接

使用mysql二进制方式连接

您可以使用MySQL二进制方式进入到mysql命令提示符下来连接MySQL数据库。

实例

以下是从命令行中连接mysql服务器的简单实例：

```
[root@host]# mysql -u root -p
Enter password:******
```

在登录成功后会出现 mysql> 命令提示窗口，你可以在上面执行任何 SQL 语句。

以上命令执行后，登录成功输出结果如下:

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2854760 to server version: 5.0.9

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
```

在以上实例中，我们使用了root用户登录到mysql服务器，当然你也可以使用其他mysql用户登录。

如果用户权限足够，任何用户都可以在mysql的命令提示窗口中进行SQL操作。

退出 mysql> 命令提示窗口可以使用 exit 命令，如下所示：

```
mysql> exit
Bye
```

使用 PHP 脚本连接 MySQL

PHP 提供了 mysql_connect() 函数来连接数据库。

该函数有5个参数，在成功链接到MySQL后返回连接标识，失败返回 FALSE 。

语法

```
connection mysql_connect(server,user,passwd,new_link,client_flag);
```

参数说明：

参数	描述
server	可选。规定要连接的服务器。 可以包括端口号，例如 "hostname:port"，或者到本地套接字的路径，例如对于 localhost 的 ":/path/to/socket"。 如果 PHP 指令 mysql.default_host 未定义（默认情况），则默认值是 'localhost:3306'。
user	可选。用户名。默认值是服务器进程所有者的用户名。
passwd	可选。密码。默认值是空密码。
new_link	可选。如果用同样的参数第二次调用 mysql_connect()，将不会建立新连接，而将返回已经打开的连接标识。参数 new_link 改变此行为并使 mysql_connect() 总是打开新的连接，甚至当 mysql_connect() 曾在前面被用同样的参数调用过。
client_flag	可选。client_flags 参数可以是以下常量的组合： MYSQL_CLIENT_SSL - 使用 SSL 加密 MYSQL_CLIENT_COMPRESS - 使用压缩协议 MYSQL_CLIENT_IGNORE_SPACE - 允许函数名后的间隔 MYSQL_CLIENT_INTERACTIVE - 允许关闭连接之前的交互超时非活动时间

你可以使用PHP的 mysql_close() 函数来断开与MySQL数据库的链接。

该函数只有一个参数为mysql_connect()函数创建连接成功后返回的 MySQL 连接标识符。

语法

```
bool mysql_close ( resource $link_identifier );
```

本函数关闭指定的连接标识所关联的到 MySQL 服务器的非持久连接。如果没有指定 link_identifier，则关闭上一个打开的连接。

提示：通常不需要使用 mysql_close()，因为已打开的非持久连接会在脚本执行完毕后自动关闭。

注释：mysql_close() 不会关闭由 mysql_pconnect() 建立的持久连接。

实例

你可以尝试以下实例来连接到你的 MySQL 服务器:

```
<html>
<head>
<title>Connecting MySQL Server</title>
</head>
<body>
<?php
   $dbhost = 'localhost:3036';  //mysql服务器主机地址
   $dbuser = 'guest';      //mysql用户名
   $dbpass = 'guest123';//mysql用户名密码
   $conn = mysql_connect($dbhost, $dbuser, $dbpass);
   if(! $conn )
   {
     die('Could not connect: ' . mysql_error());
   }
   echo 'Connected successfully';
   mysql_close($conn);
?>
</body>
</html>
```

MySQL 创建数据库

使用 mysqladmin 创建数据库

使用普通用户，你可能需要特定的权限来创建或者删除 MySQL 数据库。

所以我们这边使用root用户登录，root用户拥有最高权限，可以使用 mysql mysqladmin 命令来创建数据库。

实例

以下命令简单的演示了创建数据库的过程，数据名为 TUTORIALS:

```
[root@host]# mysqladmin -u root -p create TUTORIALS
Enter password:******
```

以上命令执行成功后会创建 MySQL 数据库 TUTORIALS。

使用 PHP脚本 创建数据库

PHP使用 mysql_query 函数来创建或者删除 MySQL 数据库。

该函数有两个参数，在执行成功时返回 TRUE，否则返回 FALSE。

语法

```
bool mysql_query( sql, connection );
```

参数	描述
sql	必需。规定要发送的 SQL 查询。注释：查询字符串不应以分号结束。
connection	可选。规定 SQL 连接标识符。如果未规定，则使用上一个打开的连接。

实例

以下实例演示了使用PHP来创建一个数据库：

```
<html>
<head>
<title>Creating MySQL Database</title>
</head>
<body>
<?php
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('Could not connect: ' . mysql_error());
}
echo 'Connected successfully<br />';
$sql = 'CREATE DATABASE TUTORIALS';
$retval = mysql_query( $sql, $conn );
if(! $retval )
{
  die('Could not create database: ' . mysql_error());
}
echo "Database TUTORIALS created successfully\n";
mysql_close($conn);
?>
</body>
</html>
```

MySQL 删除数据库

使用 mysqladmin 删除数据库

使用普通用户登陆mysql服务器，你可能需要特定的权限来创建或者删除 MySQL 数据库。

所以我们这边使用root用户登录，root用户拥有最高权限，可以使用 mysql mysqladmin 命令来创建数据库。

在删除数据库过程中，务必要十分谨慎，因为在执行删除命令后，所有数据将会消失。

以下实例删除数据库TUTORIALS(该数据库在前一章节已创建)：

```
[root@host]# mysqladmin -u root -p drop TUTORIALS
Enter password:******
```

执行以上删除数据库命令后，会出现一个提示框，来确认是否真的删除数据库：

```
Dropping the database is potentially a very bad thing to do.
Any data stored in the database will be destroyed.

Do you really want to drop the 'TUTORIALS' database [y/N] y
Database "TUTORIALS" dropped
```

使用PHP脚本删除数据库

PHP使用 mysql_query 函数来创建或者删除 MySQL 数据库。

该函数有两个参数，在执行成功时返回 TRUE，否则返回 FALSE。

语法

```
bool mysql_query( sql, connection );
```

参数	描述
sql	必需。规定要发送的 SQL 查询。注释：查询字符串不应以分号结束。
connection	可选。规定 SQL 连接标识符。如果未规定，则使用上一个打开的连接。

实例

以下实例演示了使用PHP mysql_query函数来删除数据库：

```
<html>
<head>
<title>Deleting MySQL Database</title>
</head>
<body>
<?php
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('连接失败: ' . mysql_error());
}
echo '连接成功<br />';
$sql = 'DROP DATABASE TUTORIALS';
$retval = mysql_query( $sql, $conn );
if(! $retval )
{
  die('删除数据库失败: ' . mysql_error());
}
echo "数据库 TUTORIALS 删除成功\n";
mysql_close($conn);
?>
</body>
</html>
```

注意：在使用PHP脚本删除数据库时，不会出现确认是否删除信息，会直接删除指定数据库，所以你在删除数据库时要特别小心。

MySQL 选择数据库

在你连接到 MySQL 数据库后，可能有多个可以操作的数据库，所以你需要选择你要操作的数据库。

从命令提示窗口中选择MySQL数据库

在 mysql> 提示窗口中可以很简单的选择特定的数据库。你可以使用SQL命令来选择指定的数据库。

实例

以下实例选取了数据库 TUTORIALS:

```
[root@host]# mysql -u root -p
Enter password:******
mysql> use TUTORIALS;
Database changed
mysql>
```

执行以上命令后，你就已经成功选择了 TUTORIALS 数据库，在后续的操作中都会在 TUTORIALS 数据库中执行。

注意:所有的数据库名，表名，表字段都是区分大小写的。所以你在使用SQL命令时需要输入正确的名称。

使用PHP脚本选择MySQL数据库

PHP 提供了函数 mysql_select_db 来选取一个数据库。函数在执行成功后返回 TRUE ，否则返回 FALSE 。

语法

```
bool mysql_select_db( db_name, connection );
```

参数	描述
db_name	必需。规定要选择的数据库。
connection	可选。规定 MySQL 连接。如果未指定，则使用上一个连接。

实例

以下实例展示了如何使用 mysql_select_db 函数来选取一个数据库：

```
<html>
<head>
<title>Selecting MySQL Database</title>
</head>
<body>
<?php
$dbhost = 'localhost:3036';
$dbuser = 'guest';
$dbpass = 'guest123';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('Could not connect: ' . mysql_error());
}
echo 'Connected successfully';
mysql_select_db( 'TUTORIALS' );
mysql_close($conn);
?>
</body>
</html>
```

MySQL 数据类型

MySQL中定义数据字段的类型对你数据库的优化是非常重要的。

MySQL支持多种类型，大致可以分为三类：数值、日期/时间和字符串(字符)类型。

数值类型

MySQL支持所有标准SQL数值数据类型。

这些类型包括严格数值数据类型(INTEGER、SMALLINT、DECIMAL和NUMERIC)，以及近似数值数据类型(FLOAT、REAL和DOUBLE PRECISION)。

关键字INT是INTEGER的同义词，关键字DEC是DECIMAL的同义词。

BIT数据类型保存位字段值，并且支持MyISAM、MEMORY、InnoDB和BDB表。

作为SQL标准的扩展，MySQL也支持整数类型TINYINT、MEDIUMINT和BIGINT。下面的表显示了需要的每个整数类型的存储和范围。

类型	大小	范围（有符号）	范围（无符号）	用途
TINYINT	1 字节	(-128，127)	(0，255)	小整数值
SMALLINT	2 字节	(-32 768，32 767)	(0，65 535)	大整数值
MEDIUMINT	3 字节	(-8 388 608，8 388 607)	(0，16 777 215)	大整数值
INT或INTEGER	4 字节	(-2 147 483 648，2 147 483 647)	(0，4 294 967 295)	大整数值
BIGINT	8 字节	(-9 233 372 036 854 775 808，9 223 372 036 854 775 807)	(0，18 446 744 073 709 551 615)	极大整数值

FLOAT	4 字节	(-3.402 823 466 E+38，1.175 494 351 E-38)，0，(1.175 494 351 E-38，3.402 823 466 351 E+38)	0，(1.175 494 351 E-38，3.402 823 466 E+38)	单精度浮点数值
DOUBLE	8 字节	(1.797 693 134 862 315 7 E+308，2.225 073 858 507 201 4 E-308)，0，(2.225 073 858 507 201 4 E-308，1.797 693 134 862 315 7 E+308)	0，(2.225 073 858 507 201 4 E-308，1.797 693 134 862 315 7 E+308)	双精度浮点数值
DECIMAL	对DECIMAL(M,D)，如果M>D，为M+2否则为D+2	依赖于M和D的值	依赖于M和D的值	小数值

日期和时间类型

表示时间值的日期和时间类型为DATETIME、DATE、TIMESTAMP、TIME和YEAR。

每个时间类型有一个有效值范围和一个"零"值，当指定不合法的MySQL不能表示的值时使用"零"值。

TIMESTAMP类型有专有的自动更新特性，将在后面描述。

类型	大小(字节)	范围	格式	用途
DATE	3	1000-01-01/9999-12-31	YYYY-MM-DD	日期值
TIME	3	'-838:59:59'/'838:59:59'	HH:MM:SS	时间值或持续时间
YEAR	1	1901/2155	YYYY	年份值
DATETIME	8	1000-01-01 00:00:00/9999-12-31 23:59:59	YYYY-MM-DD HH:MM:SS	混合日期和时间值
TIMESTAMP	8	1970-01-01 00:00:00/2037 年某时	YYYYMMDD HHMMSS	混合日期和时间值，时间戳

字符串类型

字符串类型指CHAR、VARCHAR、BINARY、VARBINARY、BLOB、TEXT、ENUM和SET。该节描述了这些类型如何工作以及如何在查询中使用这些类型。

类型	大小	用途
CHAR	0-255字节	定长字符串
VARCHAR	0-255字节	变长字符串
TINYBLOB	0-255字节	不超过 255 个字符的二进制字符串
TINYTEXT	0-255字节	短文本字符串
BLOB	0-65 535字节	二进制形式的长文本数据
TEXT	0-65 535字节	长文本数据
MEDIUMBLOB	0-16 777 215字节	二进制形式的中等长度文本数据
MEDIUMTEXT	0-16 777 215字节	中等长度文本数据
LOGNGBLOB	0-4 294 967 295字节	二进制形式的极大文本数据
LONGTEXT	0-4 294 967 295字节	极大文本数据

CHAR和VARCHAR类型类似，但它们保存和检索的方式不同。它们的最大长度和是否尾部空格被保留等方面也不同。在存储或检索过程中不进行大小写转换。

BINARY和VARBINARY类类似于CHAR和VARCHAR，不同的是它们包含二进制字符串而不要非二进制字符串。也就是说，它们包含字节字符串而不是字符字符串。这说明它们没有字符集，并且排序和比较基于列值字节的数值值。

BLOB是一个二进制大对象，可以容纳可变数量的数据。有4种BLOB类型：TINYBLOB、BLOB、MEDIUMBLOB和LONGBLOB。它们只是可容纳值的最大长度不同。

有4种TEXT类型：TINYTEXT、TEXT、MEDIUMTEXT和LONGTEXT。这些对应4种BLOB类型，有相同的最大长度和存储需求。

MySQL 创建数据表

创建MySQL数据表需要以下信息：

- 表名
- 表字段名
- 定义每个表字段

语法

以下为创建MySQL数据表的SQL通用语法：

```
CREATE TABLE table_name (column_name column_type);
```

以下例子中我们将在 TUTORIALS 数据库中创建数据表tutorials_tbl：

```
tutorials_tbl(
   tutorial_id INT NOT NULL AUTO_INCREMENT,
   tutorial_title VARCHAR(100) NOT NULL,
   tutorial_author VARCHAR(40) NOT NULL,
   submission_date DATE,
   PRIMARY KEY ( tutorial_id )
);
```

实例解析：

- 如果你不想字段为 **NULL** 可以设置字段的属性为 **NOT NULL**， 在操作数据库时如果输入该字段的数据为**NULL** ，就会报错。
- AUTO_INCREMENT定义列为自增的属性，一般用于主键，数值会自动加1。
- PRIMARY KEY关键字用于定义列为主键。 您可以使用多列来定义主键，列间以逗号分隔。

通过命令提示符创建表

通过 mysql> 命令窗口可以很简单的创建MySQL数据表。你可以使用 SQL 语句 **CREATE TABLE** 来创建数据表。

实例

以下为创建数据表 tutorials_tbl 实例:

```
root@host# mysql -u root -p
Enter password:*******
mysql> use TUTORIALS;
Database changed
mysql> CREATE TABLE tutorials_tbl(
   -> tutorial_id INT NOT NULL AUTO_INCREMENT,
   -> tutorial_title VARCHAR(100) NOT NULL,
   -> tutorial_author VARCHAR(40) NOT NULL,
   -> submission_date DATE,
   -> PRIMARY KEY ( tutorial_id )
   -> );
Query OK, 0 rows affected (0.16 sec)
mysql>
```

注意：MySQL命令终止符为分号 (;) 。

使用PHP脚本创建数据表

你可以使用PHP的 mysql_query() 函数来创建已存在数据库的数据表。

该函数有两个参数，在执行成功时返回 TRUE，否则返回 FALSE。

语法

```
bool mysql_query( sql, connection );
```

参数	描述
sql	必需。规定要发送的 SQL 查询。注释：查询字符串不应以分号结束。
connection	可选。规定 SQL 连接标识符。如果未规定，则使用上一个打开的连接。

实例

以下实例使用了PHP脚本来创建数据表：

```
<html>
<head>
<title>Creating MySQL Tables</title>
</head>
<body>
<?php
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('Could not connect: ' . mysql_error());
}
echo 'Connected successfully<br />';
$sql = "CREATE TABLE tutorials_tbl( ".
       "tutorial_id INT NOT NULL AUTO_INCREMENT, ".
       "tutorial_title VARCHAR(100) NOT NULL, ".
       "tutorial_author VARCHAR(40) NOT NULL, ".
       "submission_date DATE, ".
       "PRIMARY KEY ( tutorial_id )); ";
mysql_select_db( 'TUTORIALS' );
$retval = mysql_query( $sql, $conn );
if(! $retval )
{
  die('Could not create table: ' . mysql_error());
}
echo "Table created successfully\n";
mysql_close($conn);
?>
</body>
</html>
```

MySQL 删除数据表

MySQL中删除数据表是非常容易操作的， 但是你再进行删除表操作时要非常小心，因为执行删除命令后所有数据都会消失。

语法

以下为删除MySQL数据表的通用语法：

```
DROP TABLE table_name ;
```

在命令提示窗口中删除数据表

在mysql>命令提示窗口中删除数据表SQL语句为 **DROP TABLE** ：

实例

以下实例删除了数据表tutorials_tbl:

```
root@host# mysql -u root -p
Enter password:*******
mysql> use TUTORIALS;
Database changed
mysql> DROP TABLE tutorials_tbl
Query OK, 0 rows affected (0.8 sec)
mysql>
```

使用PHP脚本删除数据表

PHP使用 mysql_query 函数来删除 MySQL 数据表。

该函数有两个参数，在执行成功时返回 TRUE，否则返回 FALSE。

h3> 语法

```
bool mysql_query( sql, connection );
```

参数	描述
sql	必需。规定要发送的 SQL 查询。注释：查询字符串不应以分号结束。
connection	可选。规定 SQL 连接标识符。如果未规定，则使用上一个打开的连接。

实例

以下实例使用了PHP脚本删除数据表tutorials_tbl:

```
<html>
<head>
<title>Creating MySQL Tables</title>
</head>
<body>
<?php
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('Could not connect: ' . mysql_error());
}
echo 'Connected successfully<br />';
$sql = "DROP TABLE tutorials_tbl";
mysql_select_db( 'TUTORIALS' );
$retval = mysql_query( $sql, $conn );
if(! $retval )
{
  die('Could not delete table: ' . mysql_error());
}
echo "Table deleted successfully\n";
mysql_close($conn);
?>
</body>
</html>
```

MySQL 插入数据

MySQL 表中使用 **INSERT INTO** SQL语句来插入数据。

你可以通过 mysql> 命令提示窗口中向数据表中插入数据，或者通过PHP脚本来插入数据。

语法

以下为向MySQL数据表插入数据通用的 **INSERT INTO** SQL语法：

```
INSERT INTO table_name ( field1, field2,...fieldN )
                       VALUES
                       ( value1, value2,...valueN );
```

如果数据是字符型，必须使用单引号或者双引号，如："value"。

通过命令提示窗口插入数据

以下我们将使用 SQL **INSERT INTO** 语句向 MySQL 数据表 tutorials_tbl 插入数据

实例

以下实例中我们将想 tutorials_tbl 表插入三条数据:

```
root@host# mysql -u root -p password;
Enter password:*******
mysql> use TUTORIALS;
Database changed
mysql> INSERT INTO tutorials_tbl
     ->(tutorial_title, tutorial_author, submission_date)
     ->VALUES
     ->("Learn PHP", "John Poul", NOW());
Query OK, 1 row affected (0.01 sec)
mysql> INSERT INTO tutorials_tbl
     ->(tutorial_title, tutorial_author, submission_date)
     ->VALUES
     ->("Learn MySQL", "Abdul S", NOW());
Query OK, 1 row affected (0.01 sec)
mysql> INSERT INTO tutorials_tbl
     ->(tutorial_title, tutorial_author, submission_date)
     ->VALUES
     ->("JAVA Tutorial", "Sanjay", '2007-05-06');
Query OK, 1 row affected (0.01 sec)
mysql>
```

注意： 使用箭头标记(->)不是SQL语句的一部分，它仅仅表示一个新行，如果一条SQL语句太长，我们可以通过回车键来创建一个新行来编写SQL语句，SQL语句的命令结束符为分号（;）。

在以上实例中，我们并没有提供 tutorial_id 的数据，因为该字段我们在创建表的时候已经设置它为 AUTO_INCREMENT(自动增加) 属性。 所以，该字段会自动递增而不需要我们去设置。实例中 NOW() 是一个 MySQL 函数，该函数返回日期和时间。

使用**PHP**脚本插入数据

你可以使用PHP 的 mysql_query() 函数来执行 **SQL INSERT INTO**命令来插入数据。

该函数有两个参数，在执行成功时返回 TRUE，否则返回 FALSE。

语法

```
bool mysql_query( sql, connection );
```

参数	描述
sql	必需。规定要发送的 SQL 查询。注释：查询字符串不应以分号结束。
connection	可选。规定 SQL 连接标识符。如果未规定，则使用上一个打开的连接。

实例

以下实例中程序接收用户输入的三个字段数据，并插入数据表中：

```
<html>
<head>
<title>Add New Record in MySQL Database</title>
</head>
<body>
<?php
if(isset($_POST['add']))
{
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('Could not connect: ' . mysql_error());
}

if(! get_magic_quotes_gpc() )
{
   $tutorial_title = addslashes ($_POST['tutorial_title']);
   $tutorial_author = addslashes ($_POST['tutorial_author']);
}
else
{
   $tutorial_title = $_POST['tutorial_title'];
   $tutorial_author = $_POST['tutorial_author'];
}
$submission_date = $_POST['submission_date'];

$sql = "INSERT INTO tutorials_tbl ".
       "(tutorial_title,tutorial_author, submission_date) ".
       "VALUES ".
       "('$tutorial_title','$tutorial_author','$submission_date')";
mysql_select_db('TUTORIALS');
$retval = mysql_query( $sql, $conn );
if(! $retval )
{
  die('Could not enter data: ' . mysql_error());
}
```

```
echo "Entered data successfully\n";
mysql_close($conn);
}
else
{
?>
<form method="post" action="<?php $_PHP_SELF ?>">
<table width="600" border="0" cellspacing="1" cellpadding="2">
<tr>
<td width="250">Tutorial Title</td>
<td>
<input name="tutorial_title" type="text" id="tutorial_title">
</td>
</tr>
<tr>
<td width="250">Tutorial Author</td>
<td>
<input name="tutorial_author" type="text" id="tutorial_author">
</td>
</tr>
<tr>
<td width="250">Submission Date [ yyyy-mm-dd ]</td>
<td>
<input name="submission_date" type="text" id="submission_date">
</td>
</tr>
<tr>
<td width="250"> </td>
<td> </td>
</tr>
<tr>
<td width="250"> </td>
<td>
<input name="add" type="submit" id="add" value="Add Tutorial">
</td>
</tr>
</table>
</form>
<?php
}
?>
</body>
</html>
```

在我们接收用户提交的数据时，为了数据的安全性我们需要使用 get_magic_quotes_gpc() 函数来判断特殊字符的转义是否已经开启。如果这个选项为off（未开启），返回0，那么我们就必须调用addslashes 这个函数来为字符串增加转义。

义。

你也可以添加其他检查数据的方法，比如邮箱格式验证，电话号码验证，是否为整数验证等。

MySQL 查询数据

MySQL 数据库使用SQL SELECT语句来查询数据。

你可以通过 mysql> 命令提示窗口中在数据库中查询数据，或者通过PHP脚本来查询数据。

语法

以下为在MySQL数据库中查询数据通用的 SELECT 语法：

```
SELECT field1, field2,...fieldN table_name1, table_name2...
[WHERE Clause]
[OFFSET M ][LIMIT N]
```

- 查询语句中你可以使用一个或者多个表，表之间使用逗号(,)分割，并使用WHERE语句来设定查询条件。
- SELECT 命令可以读取一条或者多条记录。
- 你可以使用星号（*）来代替其他字段，SELECT语句会返回表的所有字段数据
- 你可以使用 WHERE 语句来包含任何条件。
- 你可以通过OFFSET指定SELECT语句开始查询的数据偏移量。默认情况下偏移量为0。
- 你可以使用 LIMIT 属性来设定返回的记录数。

通过命令提示符获取数据

以下实例我们将通过 SQL SELECT 命令来获取 MySQL 数据表 tutorials_tbl 的数据：

实例

以下实例将返回数据表tutorials_tbl的所有记录:

```
root@host# mysql -u root -p password;
Enter password:*******
mysql> use TUTORIALS;
Database changed
mysql> SELECT * from tutorials_tbl
+-------------+----------------+-----------------+-----------------
| tutorial_id | tutorial_title | tutorial_author | submission_date
+-------------+----------------+-----------------+-----------------
|           1 | Learn PHP      | John Poul       | 2007-05-21
|           2 | Learn MySQL    | Abdul S         | 2007-05-21
|           3 | JAVA Tutorial  | Sanjay          | 2007-05-21
+-------------+----------------+-----------------+-----------------
3 rows in set (0.01 sec)

mysql>
```

使用PHP脚本来获取数据

使用PHP函数的mysql_query()及SQL SELECT命令来获取数据。

该函数用于执行SQL命令，然后通过 PHP 函数 mysql_fetch_array() 来使用或输出所有查询的数据。

mysql_fetch_array() 函数从结果集中取得一行作为关联数组，或数字数组，或二者兼有 返回根据从结果集取得的行生成的数组，如果没有更多行则返回 false。

以下实例为从数据表 tutorials_tbl 中读取所有记录。

实例

尝试以下实例来显示数据表 tutorials_tbl 的所有记录。

```
<?php
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('Could not connect: ' . mysql_error());
}
$sql = 'SELECT tutorial_id, tutorial_title,
               tutorial_author, submission_date
        FROM tutorials_tbl';

mysql_select_db('TUTORIALS');
$retval = mysql_query( $sql, $conn );
if(! $retval )
{
  die('Could not get data: ' . mysql_error());
}
while($row = mysql_fetch_array($retval, MYSQL_ASSOC))
{
    echo "Tutorial ID :{$row['tutorial_id']}  <br> ".
         "Title: {$row['tutorial_title']} <br> ".
         "Author: {$row['tutorial_author']} <br> ".
         "Submission Date : {$row['submission_date']} <br> ".
         "--------------------------------<br>";
}
echo "Fetched data successfully\n";
mysql_close($conn);
?>
```

以上实例中，读取的每行记录赋值给变量$row，然后再打印出每个值。

注意：记住如果你需要在字符串中使用变量，请将变量置于花括号。

在上面的例子中，PHP mysql_fetch_array()函数第二个参数为MYSQL_ASSOC，设置该参数查询结果返回关联数组，你可以使用字段名称来作为数组的索引。

PHP提供了另外一个函数mysql_fetch_assoc(), 该函数从结果集中取得一行作为关联数组。 返回根据从结果集取得的行生成的关联数组，如果没有更多行，则返回false。

实例

尝试以下实例，该实例使用了mysql_fetch_assoc()函数来输出数据表tutorial_tbl的所有记录：

```
<?php
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('Could not connect: ' . mysql_error());
}
$sql = 'SELECT tutorial_id, tutorial_title,
               tutorial_author, submission_date
        FROM tutorials_tbl';

mysql_select_db('TUTORIALS');
$retval = mysql_query( $sql, $conn );
if(! $retval )
{
  die('Could not get data: ' . mysql_error());
}
while($row = mysql_fetch_assoc($retval))
{
    echo "Tutorial ID :{$row['tutorial_id']}  <br> ".
         "Title: {$row['tutorial_title']} <br> ".
         "Author: {$row['tutorial_author']} <br> ".
         "Submission Date : {$row['submission_date']} <br> ".
         "--------------------------------<br>";
}
echo "Fetched data successfully\n";
mysql_close($conn);
?>
```

你也可以使用常量 MYSQL_NUM 作为PHP mysql_fetch_array()函数的第二个参数，返回数字数组。

实例

以下实例使用MYSQL_NUM参数显示数据表tutorials_tbl的所有记录:

```
<?php
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('Could not connect: ' . mysql_error());
}
$sql = 'SELECT tutorial_id, tutorial_title,
               tutorial_author, submission_date
        FROM tutorials_tbl';

mysql_select_db('TUTORIALS');
$retval = mysql_query( $sql, $conn );
if(! $retval )
{
  die('Could not get data: ' . mysql_error());
}
while($row = mysql_fetch_array($retval, MYSQL_NUM))
{
    echo "Tutorial ID :{$row[0]}  <br> ".
         "Title: {$row[1]} <br> ".
         "Author: {$row[2]} <br> ".
         "Submission Date : {$row[3]} <br> ".
         "--------------------------------<br>";
}
echo "Fetched data successfully\n";
mysql_close($conn);
?>
```

以上三个实例输出结果都一样。

内存释放

在我们执行完SELECT语句后，释放游标内存是一个很好的习惯。 。可以通过PHP函数mysql_free_result()来实现内存的释放。

以下实例演示了该函数的使用方法。

实例

尝试以下实例:

```
<?php
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('Could not connect: ' . mysql_error());
}
$sql = 'SELECT tutorial_id, tutorial_title,
               tutorial_author, submission_date
        FROM tutorials_tbl';

mysql_select_db('TUTORIALS');
$retval = mysql_query( $sql, $conn );
if(! $retval )
{
  die('Could not get data: ' . mysql_error());
}
while($row = mysql_fetch_array($retval, MYSQL_NUM))
{
    echo "Tutorial ID :{$row[0]}  <br> ".
         "Title: {$row[1]} <br> ".
         "Author: {$row[2]} <br> ".
         "Submission Date : {$row[3]} <br> ".
         "--------------------------------<br>";
}
mysql_free_result($retval);
echo "Fetched data successfully\n";
mysql_close($conn);
?>
```

MySQL where 子句

我们知道从MySQL表中使用SQL SELECT 语句来读取数据。

如需有条件地从表中选取数据，可将 WHERE 子句添加到 SELECT 语句中。

语法

以下是SQL SELECT 语句使用 WHERE 子句从数据表中读取数据的通用语法：

```
SELECT field1, field2,...fieldN table_name1, table_name2...
[WHERE condition1 [AND [OR]] condition2.....
```

- 查询语句中你可以使用一个或者多个表，表之间使用逗号(,)分割，并使用WHERE语句来设定查询条件。
- 你可以在WHERE子句中指定任何条件。
- 你可以使用AND或者OR指定一个或多个条件。
- WHERE子句也可以运用于SQL的 DELETE 或者 UPDATE 命令。
- WHERE 子句类似于程序语言中的if条件，根据 MySQL 表中的字段值来读取指定的数据。

以下为操作符列表，可用于 WHERE 子句中。

下表中实例假定 A为10 B为20

操作符	描述	实例
=	等号，检测两个值是否相等，如果相等返回true	(A = B) 返回false。
<>, !=	不等于，检测两个值是否相等，如果不相等返回true	(A != B) 返回 true。
>	大于号，检测左边的值是否大于右边的值, 如果左边的值大于右边的值返回true	(A > B) 返回false。
<	小于号，检测左边的值是否小于右边的值, 如果左边的值小于右边的值返回true	(A < B) 返回 true。
>=	大于等于号，检测左边的值是否大于或等于右边的值, 如果左边的值大于或等于右边的值返回true	(A >= B) 返回 false。
<=	小于等于号，检测左边的值是否小于于或等于右边的值, 如果左边的值小于或等于右边的值返回true	(A <= B) 返回 true。

如果我们想再MySQL数据表中读取指定的数据，WHERE 子句是非常有用的。

使用主键来作为 WHERE 子句的条件查询是非常快速的。

如果给定的条件在表中没有任何匹配的记录，那么查询不会返回任何数据。

从命令提示符中读取数据

我们将在SQL SELECT语句使用WHERE子句来读取MySQL数据表 tutorials_tbl 中的数据：

实例

以下实例将读取 tutorials_tbl 表中 tutorial_author 字段值为 Sanjay 的所有记录：

```
root@host# mysql -u root -p password;
Enter password:*******
mysql> use TUTORIALS;
Database changed
mysql> SELECT * from tutorials_tbl WHERE tutorial_author='Sanjay';
+-------------+----------------+-----------------+-----------------
| tutorial_id | tutorial_title | tutorial_author | submission_date
+-------------+----------------+-----------------+-----------------
|           3 | JAVA Tutorial  | Sanjay          | 2007-05-21
+-------------+----------------+-----------------+-----------------
1 rows in set (0.01 sec)

mysql>
```

除非你使用 LIKE 来比较字符串，否则MySQL的WHERE子句的字符串比较是不区分大小写的。 你可以使用 BINARY 关键字来设定WHERE子句的字符串比较是区分大小写的。

如下实例

```
root@host# mysql -u root -p password;
Enter password:*******
mysql> use TUTORIALS;
Database changed
mysql> SELECT * from tutorials_tbl \
          WHERE BINARY tutorial_author='sanjay';
Empty set (0.02 sec)

mysql>
```

使用PHP脚本读取数据

你可以使用PHP函数的mysql_query()及相同的SQL SELECT 带上 WHERE 子句的命令来获取数据。

该函数用于执行SQL命令，然后通过 PHP 函数 mysql_fetch_array() 来输出所有查询的数据。

实例

以下实例将从 tutorials_tbl 表中返回使用 tutorial_author 字段值为 Sanjay 的记录：

```
<?php
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('Could not connect: ' . mysql_error());
}
$sql = 'SELECT tutorial_id, tutorial_title,
               tutorial_author, submission_date
        FROM tutorials_tbl
        WHERE tutorial_author="Sanjay"';

mysql_select_db('TUTORIALS');
$retval = mysql_query( $sql, $conn );
if(! $retval )
{
  die('Could not get data: ' . mysql_error());
}
while($row = mysql_fetch_array($retval, MYSQL_ASSOC))
{
    echo "Tutorial ID :{$row['tutorial_id']}  <br> ".
         "Title: {$row['tutorial_title']} <br> ".
         "Author: {$row['tutorial_author']} <br> ".
         "Submission Date : {$row['submission_date']} <br> ".
         "--------------------------------<br>";
}
echo "Fetched data successfully\n";
mysql_close($conn);
?>
```

MySQL UPDATE 查询

如果我们需要修改或更新MySQL中的数据，我们可以使用 SQL UPDATE 命令来操作。.

语法

以下是 UPDATE 命令修改 MySQL 数据表数据的通用SQL语法：

```
UPDATE table_name SET field1=new-value1, field2=new-value2
[WHERE Clause]
```

- 你可以同时更新一个或多个字段。
- 你可以在 WHERE 子句中指定任何条件。
- 你可以在一个单独表中同时更新数据。

当你需要更新数据表中指定行的数据时 WHERE 子句是非常有用的。

通过命令提示符更新数据

以下我们将在 SQL UPDATE 命令使用 WHERE子句来更新tutorials_tbl表中指定的数据：

实例

以下实例将更新数据表中 tutorial_id 为 3 的 tutorial_title 字段值：

```
root@host# mysql -u root -p password;
Enter password:*******
mysql> use TUTORIALS;
Database changed
mysql> UPDATE tutorials_tbl
    -> SET tutorial_title='Learning JAVA'
    -> WHERE tutorial_id=3;
Query OK, 1 row affected (0.04 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql>
```

使用PHP脚本更新数据

PHP中使用函数mysql_query()来执行SQL语句，你可以在SQL UPDATE语句中使用或者不适用WHERE子句。

该函数与在mysql>命令提示符中执行SQL语句的效果是一样的。

实例

以下实例将更新 tutorial_id 为3的 tutorial_title 字段的数据。

```
<?php
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('Could not connect: ' . mysql_error());
}
$sql = 'UPDATE tutorials_tbl
        SET tutorial_title="Learning JAVA"
        WHERE tutorial_id=3';

mysql_select_db('TUTORIALS');
$retval = mysql_query( $sql, $conn );
if(! $retval )
{
  die('Could not update data: ' . mysql_error());
}
echo "Updated data successfully\n";
mysql_close($conn);
?>
```

MySQL DELETE 语句

你可以使用 SQL 的 DELETE FROM 命令来删除 MySQL 数据表中的记录。

你可以在mysql>命令提示符或PHP脚本中执行该命令。

语法

以下是SQL DELETE 语句从MySQL数据表中删除数据的通用语法：

```
DELETE FROM table_name [WHERE Clause]
```

- 如果没有指定 WHERE 子句，MySQL表中的所有记录将被删除。
- 你可以在 WHERE 子句中指定任何条件
- 您可以在单个表中一次性删除记录。

当你想删除数据表中指定的记录时 WHERE 子句是非常有用的。

从命令行中删除数据

这里我们将在 SQL DELETE 命令中使用 WHERE 子句来删除MySQL数据表 tutorials_tbl所选的数据。

实例

以下实例将删除 tutorial_tbl 表中 tutorial_id 为3 的记录：

```
root@host# mysql -u root -p password;
Enter password:*******
mysql> use TUTORIALS;
Database changed
mysql> DELETE FROM tutorials_tbl WHERE tutorial_id=3;
Query OK, 1 row affected (0.23 sec)

mysql>
```

使用 PHP 脚本删除数据

PHP使用 mysql_query() 函数来执行SQL语句， 你可以在SQL DELETE命令中使用或不使用 WHERE 子句。

该函数与 mysql>命令符执行SQL命令的效果是一样的。

实例

以下PHP实例将删除tutorial_tbl表中tutorial_id为3的记录:

```
<?php
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('Could not connect: ' . mysql_error());
}
$sql = 'DELETE FROM tutorials_tbl
        WHERE tutorial_id=3';

mysql_select_db('TUTORIALS');
$retval = mysql_query( $sql, $conn );
if(! $retval )
{
  die('Could not delete data: ' . mysql_error());
}
echo "Deleted data successfully\n";
mysql_close($conn);
?>
```

MySQL LIKE 子句

我们知道在MySQL中使用 SQL SELECT 命令来读取数据， 同时我们可以在 SELECT 语句中使用 WHERE 子句来获取指定的记录。

WHERE 子句中可以使用等号 (=) 来设定获取数据的条件，如 "tutorial_author = 'Sanjay'"。

但是有时候我们需要获取 tutorial_author 字段含有 "jay" 字符的所有记录，这时我们就需要在 WHERE 子句中使用 SQL LIKE 子句。

SQL LIKE 子句中使用百分号(%)字符来表示任意字符，类似于UNIX或正则表达式中的星号 (*)。

如果没有使用百分号(%), LIKE 子句与等号（=）的效果是一样的。

语法

以下是SQL SELECT 语句使用 LIKE 子句从数据表中读取数据的通用语法：

```
SELECT field1, field2,...fieldN table_name1, table_name2...
WHERE field1 LIKE condition1 [AND [OR]] filed2 = 'somevalue'
```

- 你可以在WHERE子句中指定任何条件。
- 你可以在WHERE子句中使用LIKE子句。
- 你可以使用LIKE子句代替等号(=)。
- LIKE 通常与 % 一同使用，类似于一个元字符的搜索。
- 你可以使用AND或者OR指定一个或多个条件。
- 你可以在 DELETE 或 UPDATE 命令中使用 WHERE...LIKE 子句来指定条件。

在命令提示符中使用 LIKE 子句

以下我们将在 SQL SELECT 命令中使用 WHERE...LIKE 子句来从MySQL数据表 tutorials_tbl 中读取数据。

实例

以下是我们将tutorials_tbl表中获取tutorial_author字段中以"jay"为结尾的的所有记录：

```
root@host# mysql -u root -p password;
Enter password:*******
mysql> use TUTORIALS;
Database changed
mysql> SELECT * from tutorials_tbl
    -> WHERE tutorial_author LIKE '%jay';
+-------------+----------------+-----------------+----------------
| tutorial_id | tutorial_title | tutorial_author | submission_date
+-------------+----------------+-----------------+----------------
|           3 | JAVA Tutorial  | Sanjay          | 2007-05-21
+-------------+----------------+-----------------+----------------
1 rows in set (0.01 sec)

mysql>
```

在PHP脚本中使用 LIKE 子句

你可以使用PHP函数的mysql_query()及相同的SQL SELECT 带上 WHERE...LIKE 子句的命令来获取数据。

该函数用于执行SQL命令，然后通过 PHP 函数 mysql_fetch_array() 来输出所有查询的数据。

但是如果是DELETE或者UPDATE中使用 WHERE...LIKE 子句的SQL语句，则无需使用mysql_fetch_array() 函数。

实例

以下是我们使用PHP脚本在tutorials_tbl表中读取tutorial_author字段中以"jay"为结尾的的所有记录：

```
<?php
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('Could not connect: ' . mysql_error());
}
$sql = 'SELECT tutorial_id, tutorial_title,
               tutorial_author, submission_date
        FROM tutorials_tbl
        WHERE tutorial_author LIKE "%jay%"';

mysql_select_db('TUTORIALS');
$retval = mysql_query( $sql, $conn );
if(! $retval )
{
  die('Could not get data: ' . mysql_error());
}
while($row = mysql_fetch_array($retval, MYSQL_ASSOC))
{
    echo "Tutorial ID :{$row['tutorial_id']}  <br> ".
         "Title: {$row['tutorial_title']} <br> ".
         "Author: {$row['tutorial_author']} <br> ".
         "Submission Date : {$row['submission_date']} <br> ".
         "--------------------------------<br>";
}
echo "Fetched data successfully\n";
mysql_close($conn);
?>
```

MySQL 排序

我们知道从MySQL表中使用SQL SELECT 语句来读取数据。

如果我们需要对读取的数据进行排序，我们就可以使用MySQL的 ORDER BY 子句来设定你想按哪个字段哪中方式来进行排序，再返回搜索结果。

语法

以下是SQL SELECT 语句使用 ORDER BY 子句将查询数据排序后再返回数据：

```
SELECT field1, field2,...fieldN table_name1, table_name2...
ORDER BY field1, [field2...] [ASC [DESC]]
```

- 你可以使用任何字段来作为排序的条件，从而返回排序后的查询结果。
- 你可以设定多个字段来排序。
- 你可以使用 ASC 或 DESC 关键字来设置查询结果是按升序或降序排列。 默认情况下，它是按升排列。
- 你可以添加 WHERE...LIKE 子句来设置条件。

在命令提示符中使用 **ORDER BY** 子句

以下将在 SQL SELECT 语句中使用 ORDER BY 子句来读取MySQL 数据表 tutorials_tbl 中的数据：

实例

尝试以下实例，结果将按升序排列

```
root@host# mysql -u root -p password;
Enter password:*******
mysql> use TUTORIALS;
Database changed
mysql> SELECT * from tutorials_tbl ORDER BY tutorial_author ASC
+-------------+----------------+-----------------+-----------------
| tutorial_id | tutorial_title | tutorial_author | submission_date
+-------------+----------------+-----------------+-----------------
|           2 | Learn MySQL    | Abdul S         | 2007-05-24
|           1 | Learn PHP      | John Poul       | 2007-05-24
|           3 | JAVA Tutorial  | Sanjay          | 2007-05-06
+-------------+----------------+-----------------+-----------------
3 rows in set (0.42 sec)

mysql>
```

读取 tutorials_tbl 表中所有数据并按 tutorial_author 字段的升序排列。

在PHP脚本中使用 ORDER BY 子句

你可以使用PHP函数的mysql_query()及相同的SQL SELECT 带上 ORDER BY 子句的命令来获取数据。 该函数用于执行SQL命令，然后通过 PHP 函数 mysql_fetch_array() 来输出所有查询的数据。

实例

尝试以下实例，查询后的数据按 tutorial_author 字段的降序排列后返回。

```
<?php
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('Could not connect: ' . mysql_error());
}
$sql = 'SELECT tutorial_id, tutorial_title,
               tutorial_author, submission_date
        FROM tutorials_tbl
        ORDER BY  tutorial_author DESC';

mysql_select_db('TUTORIALS');
$retval = mysql_query( $sql, $conn );
if(! $retval )
{
  die('Could not get data: ' . mysql_error());
}
while($row = mysql_fetch_array($retval, MYSQL_ASSOC))
{
    echo "Tutorial ID :{$row['tutorial_id']}  <br> ".
         "Title: {$row['tutorial_title']} <br> ".
         "Author: {$row['tutorial_author']} <br> ".
         "Submission Date : {$row['submission_date']} <br> ".
         "--------------------------------<br>";
}
echo "Fetched data successfully\n";
mysql_close($conn);
?>
```

Mysql Join的使用

在前几章节中，我们已经学会了如果在一张表中读取数据，这是相对简单的，但是在真正的应用中经常需要从多个数据表中读取数据。

本章节我们将向大家介绍如何使用MySQL 的 JOIN 在两个或多个表中查询数据。

你可以在SELECT, UPDATE 和 DELETE 语句中使用Mysql 的 join 来联合多表查询。

以下我们将演示MySQL LEFT JOIN 和 JOIN 的使用的不同之处。

在命令提示符中使用JOIN

我们在TUTORIALS数据库中有两张表 tcount_tbl 和 tutorials_tbl。两张数据表数据如下：

实例

尝试以下实例：

```
root@host# mysql -u root -p password;
Enter password:*******
mysql> use TUTORIALS;
Database changed
mysql> SELECT * FROM tcount_tbl;
+-----------------+----------------+
| tutorial_author | tutorial_count |
+-----------------+----------------+
| mahran          |             20 |
| mahnaz          |           NULL |
| Jen             |           NULL |
| Gill            |             20 |
| John Poul       |              1 |
| Sanjay          |              1 |
+-----------------+----------------+
6 rows in set (0.01 sec)
mysql> SELECT * from tutorials_tbl;
+-------------+----------------+-----------------+----------------
| tutorial_id | tutorial_title | tutorial_author | submission_date
+-------------+----------------+-----------------+----------------
|           1 | Learn PHP      | John Poul       | 2007-05-24
|           2 | Learn MySQL    | Abdul S         | 2007-05-24
|           3 | JAVA Tutorial  | Sanjay          | 2007-05-06
+-------------+----------------+-----------------+----------------
3 rows in set (0.00 sec)
mysql>
```

接下来我们就使用MySQL的JOIN来连接以上两张表来读取tutorials_tbl表中所有tutorial_author字段在tcount_tbl表对应的tutorial_count字段值：

```
mysql> SELECT a.tutorial_id, a.tutorial_author, b.tutorial_count
    -> FROM tutorials_tbl a, tcount_tbl b
    -> WHERE a.tutorial_author = b.tutorial_author;
+-------------+-----------------+----------------+
| tutorial_id | tutorial_author | tutorial_count |
+-------------+-----------------+----------------+
|           1 | John Poul       |              1 |
|           3 | Sanjay          |              1 |
+-------------+-----------------+----------------+
2 rows in set (0.01 sec)
mysql>
```

在PHP脚本中使用JOIN

PHP 中使用mysql_query()函数来执行SQL语句，你可以使用以上的相同的SQL语句作为mysql_query()函数的参数。

尝试如下实例:

```
<?php
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('Could not connect: ' . mysql_error());
}
$sql = 'SELECT a.tutorial_id, a.tutorial_author, b.tutorial_count
        FROM tutorials_tbl a, tcount_tbl b
        WHERE a.tutorial_author = b.tutorial_author';

mysql_select_db('TUTORIALS');
$retval = mysql_query( $sql, $conn );
if(! $retval )
{
  die('Could not get data: ' . mysql_error());
}
while($row = mysql_fetch_array($retval, MYSQL_ASSOC))
{
    echo "Author:{$row['tutorial_author']}  <br> ".
         "Count: {$row['tutorial_count']} <br> ".
         "Tutorial ID: {$row['tutorial_id']} <br> ".
         "--------------------------------<br>";
}
echo "Fetched data successfully\n";
mysql_close($conn);
?>
```

MySQL LEFT JOIN

MySQL left join 与 join 有所不同。 MySQL LEFT JOIN 会读取左边数据表的全部数据，即便右边表无对应数据。

实例

尝试以下实例，理解MySQL LEFT JOIN的应用：

```
root@host# mysql -u root -p password;
Enter password:*******
mysql> use TUTORIALS;
Database changed
mysql> SELECT a.tutorial_id, a.tutorial_author, b.tutorial_count
    -> FROM tutorials_tbl a LEFT JOIN tcount_tbl b
    -> ON a.tutorial_author = b.tutorial_author;
+-------------+-----------------+----------------+
| tutorial_id | tutorial_author | tutorial_count |
+-------------+-----------------+----------------+
|           1 | John Poul       |              1 |
|           2 | Abdul S         |           NULL |
|           3 | Sanjay          |              1 |
+-------------+-----------------+----------------+
3 rows in set (0.02 sec)
```

以上实例中使用了LEFT JOIN，该语句会读取左边的数据表tutorials_tbl的所有选取的字段数据，即便在右侧表tcount_tbl中没有对应的tutorial_author字段值。

MySQL NULL 值处理

我们已经知道MySQL使用 SQL SELECT 命令及 WHERE 子句来读取数据表中的数据,但是当提供的查询条件字段为 NULL 时，该命令可能就无法正常工作。

为了处理这种情况，MySQL提供了三大运算符:

- **IS NULL:** 当列的值是NULL,此运算符返回true。
- **IS NOT NULL:** 当列的值不为NULL, 运算符返回true。
- **<=>:** 比较操作符（不同于=运算符），当比较的的两个值为NULL时返回true。

关于 NULL 的条件比较运算是比较特殊的。你不能使用 = NULL 或 != NULL 在列中查找 NULL 值 。

在MySQL中，NULL值与任何其它值的比较（即使是NULL）永远返回false，即 NULL = NULL 返回false 。

MySQL中处理NULL使用IS NULL和IS NOT NULL运算符。

在命令提示符中使用 NULL 值

以下实例中假设数据库 TUTORIALS 中的表 tcount_tbl 含有两列 tutorial_author 和 tutorial_count, tutorial_count 中设置插入NULL值。

实例

尝试以下实例:

```
root@host# mysql -u root -p password;
Enter password:*******
mysql> use TUTORIALS;
Database changed
mysql> create table tcount_tbl
    -> (
    -> tutorial_author varchar(40) NOT NULL,
    -> tutorial_count  INT
    -> );
Query OK, 0 rows affected (0.05 sec)
mysql> INSERT INTO tcount_tbl
    -> (tutorial_author, tutorial_count) values ('mahran', 20);
mysql> INSERT INTO tcount_tbl
    -> (tutorial_author, tutorial_count) values ('mahnaz', NULL);
mysql> INSERT INTO tcount_tbl
    -> (tutorial_author, tutorial_count) values ('Jen', NULL);
mysql> INSERT INTO tcount_tbl
    -> (tutorial_author, tutorial_count) values ('Gill', 20);

mysql> SELECT * from tcount_tbl;
+-----------------+----------------+
| tutorial_author | tutorial_count |
+-----------------+----------------+
| mahran          |             20 |
| mahnaz          |           NULL |
| Jen             |           NULL |
| Gill            |             20 |
+-----------------+----------------+
4 rows in set (0.00 sec)

mysql>
```

以下实例中你可以看到 = 和 != 运算符是不起作用的：

```
mysql> SELECT * FROM tcount_tbl WHERE tutorial_count = NULL;
Empty set (0.00 sec)
mysql> SELECT * FROM tcount_tbl WHERE tutorial_count != NULL;
Empty set (0.01 sec)
```

查找数据表中 tutorial_count 列是否为 NULL，必须使用IS NULL和IS NOT NULL，如下实例：

```
mysql> SELECT * FROM tcount_tbl
    -> WHERE tutorial_count IS NULL;
+-----------------+----------------+
| tutorial_author | tutorial_count |
+-----------------+----------------+
| mahnaz          |           NULL |
| Jen             |           NULL |
+-----------------+----------------+
2 rows in set (0.00 sec)
mysql> SELECT * from tcount_tbl
    -> WHERE tutorial_count IS NOT NULL;
+-----------------+----------------+
| tutorial_author | tutorial_count |
+-----------------+----------------+
| mahran          |             20 |
| Gill            |             20 |
+-----------------+----------------+
2 rows in set (0.00 sec)
```

使用PHP脚本处理 NULL 值

PHP脚本中你可以在 if...else 语句来处理变量是否为空，并生成相应的条件语句。

以下实例中PHP设置了$tutorial_count变量，然后使用该变量与数据表中的 tutorial_count 字段进行比较：

```
<?php
$dbhost = 'localhost:3036';
$dbuser = 'root';
$dbpass = 'rootpassword';
$conn = mysql_connect($dbhost, $dbuser, $dbpass);
if(! $conn )
{
  die('Could not connect: ' . mysql_error());
}
if( isset($tutorial_count ))
{
   $sql = 'SELECT tutorial_author, tutorial_count
           FROM  tcount_tbl
           WHERE tutorial_count = $tutorial_count';
}
else
{
   $sql = 'SELECT tutorial_author, tutorial_count
           FROM  tcount_tbl
           WHERE tutorial_count IS $tutorial_count';
}

mysql_select_db('TUTORIALS');
$retval = mysql_query( $sql, $conn );
if(! $retval )
{
  die('Could not get data: ' . mysql_error());
}
while($row = mysql_fetch_array($retval, MYSQL_ASSOC))
{
    echo "Author:{$row['tutorial_author']}  <br> ".
         "Count: {$row['tutorial_count']} <br> ".
         "--------------------------------<br>";
}
echo "Fetched data successfully\n";
mysql_close($conn);
?>
```

MySQL 正则表达式

在前面的章节我们已经了解到MySQL可以通过 **LIKE ...%** 来进行模糊匹配。

MySQL 同样也支持其他正则表达式的匹配， MySQL中使用 REGEXP 操作符来进行正则表达式匹配。

如果您了解PHP或Perl，那么操作起来就非常简单，因为MySQL的正则表达式匹配与这些脚本的类似。

下表中的正则模式可应用于 REGEXP 操作符中。

<table>
<tr><th>模式</th><th>描述</th></tr>
<tr><td>^</td><td>匹配输入字符串的开始位置。如果设置了 RegExp 对象的 Multiline 属性，^ 也匹配 '\n' 或 '\r' 之后的位置。</td></tr>
<tr><td>$</td><td>匹配输入字符串的结束位置。如果设置了RegExp 对象的 Multiline 属性，$ 也匹配 '\n' 或 '\r' 之前的位置。</td></tr>
<tr><td>.</td><td>匹配除 "\n" 之外的任何单个字符。要匹配包括 '\n' 在内的任何字符，请使用象 '[.\n]' 的模式。</td></tr>
<tr><td>[...]</td><td>字符集合。匹配所包含的任意一个字符。例如， '[abc]' 可以匹配 "plain" 中的 'a'。</td></tr>
<tr><td>[^...]</td><td>负值字符集合。匹配未包含的任意字符。例如， '[^abc]' 可以匹配 "plain" 中的'p'。</td></tr>
<tr><td>p1|p2|p3</td><td>匹配 p1 或 p2 或 p3。例如，'z|food' 能匹配 "z" 或 "food"。'(z|f)ood' 则匹配 "zood" 或 "food"。</td></tr>
<tr><td>*</td><td>匹配前面的子表达式零次或多次。例如，zo 能匹配 ”z” 以及 ”zoo”。 等价于{0,}。</td></tr>
<tr><td>+</td><td>匹配前面的子表达式一次或多次。例如，'zo+' 能匹配 "zo" 以及 "zoo"，但不能匹配 "z"。+ 等价于 {1,}。</td></tr>
<tr><td>{n}</td><td>n 是一个非负整数。匹配确定的 n 次。例如，'o{2}' 不能匹配 "Bob" 中的 'o'，但是能匹配 "food" 中的两个 o。</td></tr>
<tr><td>{n,m}</td><td>m 和 n 均为非负整数，其中n <= m。最少匹配 n 次且最多匹配 m 次。</td></tr>
</table>

实例

了解以上的正则需求后，我们就可以更加自己的需求来编写带有正则表达式的SQL语句。以下我们将列出几个小实例(表名：person_tbl)来加深我们的理解：

查找name字段中以'st'为开头的所有数据：

```
mysql> SELECT name FROM person_tbl WHERE name REGEXP '^st';
```

查找name字段中以'ok'为结尾的所有数据：

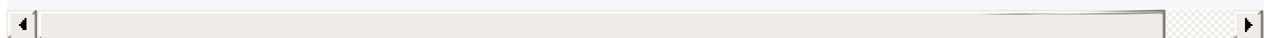
```
mysql> SELECT name FROM person_tbl WHERE name REGEXP 'ok$';
```

查找name字段中包含'mar'字符串的所有数据：

```
mysql> SELECT name FROM person_tbl WHERE name REGEXP 'mar';
```

查找name字段中以元音字符开头且以'ok'字符串结尾的所有数据：

```
mysql> SELECT name FROM person_tbl WHERE name REGEXP '^[aeiou]|ok$
```



MySQL 事务

MySQL 事务主要用于处理操作量大，复杂度高的数据。比如说，在人员管理系统中，你删除一个人员，你即需要删除人员的基本资料，也要删除和该人员相关的信息，如信箱，文章等等，这样，这些数据库操作语句就构成一个事务！

- 在MySQL中只有使用了Innodb数据库引擎的数据库或表才支持事务
- 事务处理可以用来维护数据库的完整性，保证成批的SQL语句要么全部执行，要么全部不执行
- 事务用来管理insert,update,delete语句

一般来说，事务是必须满足4个条件（ACID）：Atomicity（原子性）、Consistency（稳定性）、Isolation（隔离性）、Durability（可靠性）

- 1、事务的原子性：一组事务，要么成功；要么撤回。
- 2、稳定性：有非法数据（外键约束之类），事务撤回。
- 3、隔离性：事务独立运行。一个事务处理后的结果，影响了其他事务，那么其他事务会撤回。事务的100%隔离，需要牺牲速度。
- 4、可靠性：软、硬件崩溃后，InnoDB数据表驱动会利用日志文件重构修改。可靠性和高速度不可兼得， innodb_flush_log_at_trx_commit选项 决定什么时候吧事务保存到日志里。

在Mysql控制台使用事务来操作

1，开始一个事务

```
start transaction
```

2, 做保存点

```
save point 保存点名称
```

3, 操作

4，可以回滚，可以提交，没有问题，就提交，有问题就回滚。

PHP中使用事务实例

```
<?php
$handler=mysql_connect("localhost","root","password");
mysql_select_db("task");
mysql_query("SET AUTOCOMMIT=0");//设置为不自动提交，因为MYSQL默认立即执行
mysql_query("BEGIN");//开始事务定义
if(!mysql_query("insert into trans (id) values('2')"))
{
mysql_query("ROOLBACK");//判断当执行失败时回滚
}
if(!mysql_query("insert into trans (id) values('4')"))
{
mysql_query("ROOLBACK");//判断执行失败回滚
}
mysql_query("COMMIT");//执行事务
mysql_close($handler);
?>
```

MySQL ALTER命令

当我们需要修改数据表名或者修改数据表字段时，就需要使用到MySQL ALTER命令。

开始本章教程前让我们先创建一张表，表名为：testalter_tbl。

```
root@host# mysql -u root -p password;
Enter password:*******
mysql> use TUTORIALS;
Database changed
mysql> create table testalter_tbl
    -> (
    -> i INT,
    -> c CHAR(1)
    -> );
Query OK, 0 rows affected (0.05 sec)
mysql> SHOW COLUMNS FROM testalter_tbl;
+-------+---------+------+-----+---------+-------+
| Field | Type    | Null | Key | Default | Extra |
+-------+---------+------+-----+---------+-------+
| i     | int(11) | YES  |     | NULL    |       |
| c     | char(1) | YES  |     | NULL    |       |
+-------+---------+------+-----+---------+-------+
2 rows in set (0.00 sec)
```

删除，添加或修改表字段

如下命令使用了 ALTER 命令及 DROP 子句来删除以上创建表的 i 字段：

```
mysql> ALTER TABLE testalter_tbl  DROP i;
```

如果数据表中只剩余一个字段则无法使用DROP来删除字段。

MySQL 中使用 ADD 子句来想数据表中添加列，如下实例在表 testalter_tbl 中添加 i 字段，并定义数据类型:

```
mysql> ALTER TABLE testalter_tbl ADD i INT;
```

执行以上命令后，i 字段会自动添加到数据表字段的末尾。

```
mysql> SHOW COLUMNS FROM testalter_tbl;
+-------+---------+------+-----+---------+-------+
| Field | Type    | Null | Key | Default | Extra |
+-------+---------+------+-----+---------+-------+
| c     | char(1) | YES  |     | NULL    |       |
| i     | int(11) | YES  |     | NULL    |       |
+-------+---------+------+-----+---------+-------+
2 rows in set (0.00 sec)
```

如果你需要指定新增字段的位置，可以使用MySQL提供的关键字 FIRST (设定位第一列)， AFTER 字段名（设定位于某个字段之后）。

尝试以下 ALTER TABLE 语句, 在执行成功后，使用 SHOW COLUMNS 查看表结构的变化：

```
ALTER TABLE testalter_tbl DROP i;
ALTER TABLE testalter_tbl ADD i INT FIRST;
ALTER TABLE testalter_tbl DROP i;
ALTER TABLE testalter_tbl ADD i INT AFTER c;
```

FIRST 和 AFTER 关键字只占用于 ADD 子句，所以如果你想重置数据表字段的位置就需要先使用 DROP 删除字段然后使用 ADD 来添加字段并设置位置。

修改字段类型及名称

如果需要修改字段类型及名称, 你可以在ALTER命令中使用 MODIFY 或 CHANGE 子句 。

例如，把字段 c 的类型从 CHAR(1) 改为 CHAR(10)，可以执行以下命令:

```
mysql> ALTER TABLE testalter_tbl MODIFY c CHAR(10);
```

使用 CHANGE 子句, 语法有很大的不同。 在 CHANGE 关键字之后， 紧跟着的是你要修改的字段名，然后指定新字段的类型及名称。尝试如下实例：

```
mysql> ALTER TABLE testalter_tbl CHANGE i j BIGINT;
```

```
mysql> ALTER TABLE testalter_tbl CHANGE j j INT;
```

ALTER TABLE 对 Null 值和默认值的影响

当你修改字段时，你可以指定是否包含只或者是否设置默认值。

以下实例，指定字段 j 为 NOT NULL 且默认值为100 。

```
mysql> ALTER TABLE testalter_tbl
    -> MODIFY j BIGINT NOT NULL DEFAULT 100;
```

如果你不设置默认值，MySQL会自动设置该字段默认为 NULL。

修改字段默认值

你可以使用 ALTER 来修改字段的默认值，尝试以下实例：

```
mysql> ALTER TABLE testalter_tbl ALTER i SET DEFAULT 1000;
mysql> SHOW COLUMNS FROM testalter_tbl;
+-------+---------+------+-----+---------+-------+
| Field | Type    | Null | Key | Default | Extra |
+-------+---------+------+-----+---------+-------+
| c     | char(1) | YES  |     | NULL    |       |
| i     | int(11) | YES  |     | 1000    |       |
+-------+---------+------+-----+---------+-------+
2 rows in set (0.00 sec)
```

你也可以使用 ALTER 命令及 DROP子句来删除字段的默认值，如下实例：

```
mysql> ALTER TABLE testalter_tbl ALTER i DROP DEFAULT;
mysql> SHOW COLUMNS FROM testalter_tbl;
+-------+---------+------+-----+---------+-------+
| Field | Type    | Null | Key | Default | Extra |
+-------+---------+------+-----+---------+-------+
| c     | char(1) | YES  |     | NULL    |       |
| i     | int(11) | YES  |     | NULL    |       |
+-------+---------+------+-----+---------+-------+
2 rows in set (0.00 sec)
Changing a Table Type:
```

修改数据表类型，可以使用 ALTER 命令及 TYPE 子句来完成。尝试以下实例，我们将表 testalter_tbl 的类型修改为 MYISAM ：

注意：查看数据表类型可以使用 SHOW TABLE STATUS 语句。

```
mysql> ALTER TABLE testalter_tbl TYPE = MYISAM;
mysql>  SHOW TABLE STATUS LIKE 'testalter_tbl'\G
*************************** 1\. row ****************
           Name: testalter_tbl
           Type: MyISAM
     Row_format: Fixed
           Rows: 0
 Avg_row_length: 0
    Data_length: 0
Max_data_length: 25769803775
   Index_length: 1024
      Data_free: 0
 Auto_increment: NULL
    Create_time: 2007-06-03 08:04:36
    Update_time: 2007-06-03 08:04:36
     Check_time: NULL
 Create_options:
        Comment:
1 row in set (0.00 sec)
```

修改表名

如果需要修改数据表的名称，可以在 ALTER TABLE 语句中使用 RENAME 子句来实现。

尝试以下实例将数据表 testalter_tbl 重命名为 alter_tbl：

```
mysql> ALTER TABLE testalter_tbl RENAME TO alter_tbl;
```

ALTER 命令还可以用来创建及删除MySQL数据表的索引，该功能我们会在接下来的章节中介绍。

MySQL 索引

MySQL索引的建立对于MySQL的高效运行是很重要的，索引可以大大提高MySQL的检索速度。

打个比方，如果合理的设计且使用索引的MySQL是一辆兰博基尼的话，那么没有设计和使用索引的MySQL就是一个人力三轮车。

索引分单列索引和组合索引。单列索引，即一个索引只包含单个列，一个表可以有多个单列索引，但这不是组合索引。组合索引，即一个索包含多个列。

创建索引时，你需要确保该索引是应用在 SQL 查询语句的条件(一般作为 WHERE 子句的条件)。

实际上，索引也是一张表，该表保存了主键与索引字段，并指向实体表的记录。

上面都在说使用索引的好处，但过多的使用索引将会造成滥用。因此索引也会有它的缺点：虽然索引大大提高了查询速度，同时却会降低更新表的速度，如对表进行INSERT、UPDATE和DELETE。因为更新表时，MySQL不仅要保存数据，还要保存一下索引文件。

建立索引会占用磁盘空间的索引文件。

普通索引

创建索引

这是最基本的索引，它没有任何限制。它有以下几种创建方式：

```
CREATE INDEX indexName ON mytable(username(length));
```

如果是CHAR，VARCHAR类型，length可以小于字段实际长度；如果是BLOB和TEXT类型，必须指定 length。

修改表结构

```
ALTER mytable ADD INDEX [indexName] ON (username(length))
```

创建表的时候直接指定

```
CREATE TABLE mytable(

ID INT NOT NULL,

username VARCHAR(16) NOT NULL,

INDEX [indexName] (username(length))

);
```

删除索引的语法

```
DROP INDEX [indexName] ON mytable;
```

唯一索引

它与前面的普通索引类似，不同的就是：索引列的值必须唯一，但允许有空值。如果是组合索引，则列值的组合必须唯一。它有以下几种创建方式：

创建索引

```
CREATE UNIQUE INDEX indexName ON mytable(username(length))
```

修改表结构

```
ALTER mytable ADD UNIQUE [indexName] ON (username(length))
```

创建表的时候直接指定

```
CREATE TABLE mytable(

ID INT NOT NULL,

username VARCHAR(16) NOT NULL,

UNIQUE [indexName] (username(length))

);
```

使用ALTER 命令添加和删除索引

有四种方式来添加数据表的索引：

- **ALTER TABLE tbl_name ADD PRIMARY KEY (column_list):** 该语句添加一个主键，这意味着索引值必须是唯一的，且不能为NULL。
- **ALTER TABLE tbl_name ADD UNIQUE index_name (column_list):** 这条语句创建索引的值必须是唯一的（除了NULL外，NULL可能会出现多次）。
- **ALTER TABLE tbl_name ADD INDEX index_name (column_list):** 添加普通索引，索引值可出现多次。
- **ALTER TABLE tbl_name ADD FULLTEXT index_name (column_list):**该语句指定了索引为 FULLTEXT ，用于全文索引。

以下实例为在表中添加索引。

```
mysql> ALTER TABLE testalter_tbl ADD INDEX (c);
```

你还可以在 ALTER 命令中使用 DROP 子句来删除索引。尝试以下实例删除索引:

```
mysql> ALTER TABLE testalter_tbl DROP INDEX (c);
```

使用 ALTER 命令添加和删除主键

主键只能作用于一个列上，添加主键索引时，你需要确保该主键默认不为空（NOT NULL）。实例如下：

```
mysql> ALTER TABLE testalter_tbl MODIFY i INT NOT NULL;
mysql> ALTER TABLE testalter_tbl ADD PRIMARY KEY (i);
```

你也可以使用 ALTER 命令删除主键：

```
mysql> ALTER TABLE testalter_tbl DROP PRIMARY KEY;
```

删除指定时只需指定PRIMARY KEY，但在删除索引时，你必须知道索引名。

显示索引信息

你可以使用 SHOW INDEX 命令来列出表中的相关的索引信息。可以通过添加 \G 来格式化输出信息。

尝试以下实例:

```
mysql> SHOW INDEX FROM table_name\G
........
```

MySQL 临时表

MySQL 临时表在我们需要保存一些临时数据时是非常有用的。临时表只在当前连接可见，当关闭连接时，Mysql会自动删除表并释放所有空间。

临时表在MySQL 3.23版本中添加，如果你的MySQL版本低于 3.23版本就无法使用MySQL的临时表。不过现在一般很少有再使用这么低版本的MySQL数据库服务了。

MySQL临时表只在当前连接可见，如果你使用PHP脚本来创建MySQL临时表，那没当PHP脚本执行完成后，该临时表也会自动销毁。

如果你使用了其他MySQL客户端程序连接MySQL数据库服务器来创建临时表，那么只有在关闭客户端程序时才会销毁临时表，当然你也可以手动销毁。

实例

以下展示了使用MySQL 临时表的简单实例，以下的SQL代码可以适用于PHP脚本的mysql_query()函数。

```
mysql> CREATE TEMPORARY TABLE SalesSummary (
    -> product_name VARCHAR(50) NOT NULL
    -> , total_sales DECIMAL(12,2) NOT NULL DEFAULT 0.00
    -> , avg_unit_price DECIMAL(7,2) NOT NULL DEFAULT 0.00
    -> , total_units_sold INT UNSIGNED NOT NULL DEFAULT 0
);
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO SalesSummary
    -> (product_name, total_sales, avg_unit_price, total_units_sol
    -> VALUES
    -> ('cucumber', 100.25, 90, 2);

mysql> SELECT * FROM SalesSummary;
+--------------+-------------+----------------+------------------+
| product_name | total_sales | avg_unit_price | total_units_sold |
+--------------+-------------+----------------+------------------+
| cucumber     |      100.25 |          90.00 |                2 |
+--------------+-------------+----------------+------------------+
1 row in set (0.00 sec)
```

当你使用 **SHOW TABLES**命令显示数据表列表时，你将无法看到 SalesSummary 表。

如果你退出当前MySQL会话，再使用 **SELECT**命令来读取原先创建的临时表数据，那你会发现数据库中没有该表的存在，因为在你退出时该临时表已经被销毁了。

删除MySQL 临时表

默认情况下，当你断开与数据库的连接后，临时表就会自动被销毁。当然你也可以在当前MySQL会话使用 **DROP TABLE** 命令来手动删除临时表。

以下是手动删除临时表的实例：

```
mysql> CREATE TEMPORARY TABLE SalesSummary (
    -> product_name VARCHAR(50) NOT NULL
    -> , total_sales DECIMAL(12,2) NOT NULL DEFAULT 0.00
    -> , avg_unit_price DECIMAL(7,2) NOT NULL DEFAULT 0.00
    -> , total_units_sold INT UNSIGNED NOT NULL DEFAULT 0
);
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO SalesSummary
    -> (product_name, total_sales, avg_unit_price, total_units_sol
    -> VALUES
    -> ('cucumber', 100.25, 90, 2);

mysql> SELECT * FROM SalesSummary;
+--------------+-------------+----------------+------------------+
| product_name | total_sales | avg_unit_price | total_units_sold |
+--------------+-------------+----------------+------------------+
| cucumber     |      100.25 |          90.00 |                2 |
+--------------+-------------+----------------+------------------+
1 row in set (0.00 sec)
mysql> DROP TABLE SalesSummary;
mysql>  SELECT * FROM SalesSummary;
ERROR 1146: Table 'TUTORIALS.SalesSummary' doesn't exist
```

MySQL 复制表

如果我们需要完全的复制MySQL的数据表，包括表的结构，索引，默认值等。 如果仅仅使用**CREATE TABLE ... SELECT** 命令，是无法实现的。

本章节将为大家介绍如何完整的复制MySQL数据表，步骤如下：

- 使用 **SHOW CREATE TABLE** 命令获取创建数据表(**CREATE TABLE**) 语句，该语句包含了原数据表的结构，索引等。
- 复制以下命令显示的SQL语句，修改数据表名，并执行SQL语句，通过以上命令 将完全的复制数据表结构。
- 如果你想复制表的内容，你就可以使用 **INSERT INTO ... SELECT** 语句来实现。

实例

尝试以下实例来复制表 tutorials_tbl 。

步骤一：

获取数据表的完整结构。

```
mysql> SHOW CREATE TABLE tutorials_tbl \G;
*************************** 1\. row ***************************
       Table: tutorials_tbl
Create Table: CREATE TABLE `tutorials_tbl` (
  `tutorial_id` int(11) NOT NULL auto_increment,
  `tutorial_title` varchar(100) NOT NULL default '',
  `tutorial_author` varchar(40) NOT NULL default '',
  `submission_date` date default NULL,
  PRIMARY KEY  (`tutorial_id`),
  UNIQUE KEY `AUTHOR_INDEX` (`tutorial_author`)
) TYPE=MyISAM
1 row in set (0.00 sec)

ERROR:
No query specified
```

步骤二：

修改SQL语句的数据表名，并执行SQL语句。

```
mysql> CREATE TABLE `clone_tbl` (
  -> `tutorial_id` int(11) NOT NULL auto_increment,
  -> `tutorial_title` varchar(100) NOT NULL default '',
  -> `tutorial_author` varchar(40) NOT NULL default '',
  -> `submission_date` date default NULL,
  -> PRIMARY KEY  (`tutorial_id`),
  -> UNIQUE KEY `AUTHOR_INDEX` (`tutorial_author`)
-> ) TYPE=MyISAM;
Query OK, 0 rows affected (1.80 sec)
```

步骤三：

执行完第二步骤后，你将在数据库中创建新的克隆表 clone_tbl。 如果你想拷贝数据表的数据你可以使用 **INSERT INTO... SELECT** 语句来实现。

```
mysql> INSERT INTO clone_tbl (tutorial_id,
    ->                         tutorial_title,
    ->                         tutorial_author,
    ->                         submission_date)
    -> SELECT tutorial_id,tutorial_title,
    ->        tutorial_author,submission_date,
    -> FROM tutorials_tbl;
Query OK, 3 rows affected (0.07 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

执行以上步骤后，你将完整的复制表，包括表结构及表数据。

'Attr.nodeValue' is deprecated. Please use 'value' instead. adsbygoogle.js:32

MySQL 元数据

你可能想知道MySQL以下三种信息：

- 查询结果信息： SELECT, UPDATE 或 DELETE语句影响的记录数。
- 数据库和数据表的信息： 包含了数据库及数据表的结构信息。
- **MySQL**服务器信息： 包含了数据库服务器的当前状态，版本号等。

在MySQL的命令提示符中，我们可以很容易的获取以上服务器信息。 但如果使用Perl或PHP等脚本语言，你就需要调用特定的接口函数来获取。 接下来我们会详细介绍。

获取查询语句影响的记录数

PERL 实例

在 DBI 脚本中， 语句影响的记录数通过函数 do( ) 或 execute( )返回：

```
# 方法 1
# 使用do( ) 执行  $query
my $count = $dbh->do ($query);
# 如果发生错误会输出 0
printf "%d rows were affected\n", (defined ($count) ? $count : 0);

# 方法 2
# 使用prepare( ) 及 execute( ) 执行  $query
my $sth = $dbh->prepare ($query);
my $count = $sth->execute ( );
printf "%d rows were affected\n", (defined ($count) ? $count : 0);
```

PHP 实例

在PHP中，你可以使用 mysql_affected_rows( ) 函数来获取查询语句影响的记录数。

```
$result_id = mysql_query ($query, $conn_id);
# 如果查询失败返回
$count = ($result_id ? mysql_affected_rows ($conn_id) : 0);
print ("$count rows were affected\n");
```

数据库和数据表列表

你可以很容易的在MySQL服务器中获取数据库和数据表列表。 如果你没有足够的权限，结果将返回 null。

你也可以使用 SHOW TABLES 或 SHOW DATABASES 语句来获取数据库和数据表列表。

PERL 实例

```
# 获取当前数据库中所有可用的表。
my @tables = $dbh->tables ( );
foreach $table (@tables ){
   print "Table Name $table\n";
}
```

PHP 实例

```
<?php
$con = mysql_connect("localhost", "userid", "password");
if (!$con)
{
  die('Could not connect: ' . mysql_error());
}

$db_list = mysql_list_dbs($con);

while ($db = mysql_fetch_object($db_list))
{
  echo $db->Database . "<br />";
}
mysql_close($con);
?>
```

获取服务器元数据

以下命令语句可以在MySQL的命令提示符使用，也可以在脚本中 使用，如PHP脚本。

命令	描述
SELECT VERSION( )	服务器版本信息
SELECT DATABASE( )	当前数据库名 (或者返回空)
SELECT USER( )	当前用户名
SHOW STATUS	服务器状态
SHOW VARIABLES	服务器配置变量

MySQL 序列使用

MySQL序列是一组整数：1, 2, 3, ...，由于一张数据表只能有一个字段自增主键，如果你想实现其他字段也实现自动增加，就可以使用MySQL序列来实现。

本章我们将介绍如何使用MySQL的序列。

使用AUTO_INCREMENT

MySQL中最简单使用序列的方法就是使用 MySQL AUTO_INCREMENT 来定义列。

实例

以下实例中创建了数据表insect， insect中id无需指定值可实现自动增长。

```
mysql> CREATE TABLE insect
    -> (
    -> id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    -> PRIMARY KEY (id),
    -> name VARCHAR(30) NOT NULL, # type of insect
    -> date DATE NOT NULL, # date collected
    -> origin VARCHAR(30) NOT NULL # where collected
);
Query OK, 0 rows affected (0.02 sec)
mysql> INSERT INTO insect (id,name,date,origin) VALUES
    -> (NULL,'housefly','2001-09-10','kitchen'),
    -> (NULL,'millipede','2001-09-10','driveway'),
    -> (NULL,'grasshopper','2001-09-10','front yard');
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
mysql> SELECT * FROM insect ORDER BY id;
+----+-------------+------------+------------+
| id | name        | date       | origin     |
+----+-------------+------------+------------+
|  1 | housefly    | 2001-09-10 | kitchen    |
|  2 | millipede   | 2001-09-10 | driveway   |
|  3 | grasshopper | 2001-09-10 | front yard |
+----+-------------+------------+------------+
3 rows in set (0.00 sec)
```

获取AUTO_INCREMENT值

在MySQL的客户端中你可以使用 SQL中的LAST_INSERT_ID( ) 函数来获取最后的插入表中的自增列的值。

在PHP或PERL脚本中也提供了相应的函数来获取最后的插入表中的自增列的值。

PERL实例

使用 mysql_insertid 属性来获取 AUTO_INCREMENT 的值。 实例如下：

```
$dbh->do ("INSERT INTO insect (name,date,origin)
VALUES('moth','2001-09-14','windowsill')");
my $seq = $dbh->{mysql_insertid};
```

PHP实例

PHP 通过 mysql_insert_id ()函数来获取执行的插入SQL语句中 AUTO_INCREMENT列的值。

```
mysql_query ("INSERT INTO insect (name,date,origin)
VALUES('moth','2001-09-14','windowsill')", $conn_id);
$seq = mysql_insert_id ($conn_id);
```

重置序列

如果你删除了数据表中的多条记录，并希望对剩下数据的AUTO_INCREMENT列进行重新排列，那么你可以通过删除自增的列，然后重新添加来实现。 不过该操作要非常小心，如果在删除的同时又有新记录添加，有可能会出现数据混乱。操作如下所示：

```
mysql> ALTER TABLE insect DROP id;
mysql> ALTER TABLE insect
    -> ADD id INT UNSIGNED NOT NULL AUTO_INCREMENT FIRST,
    -> ADD PRIMARY KEY (id);
```

设置序列的开始值

一般情况下序列的开始值为1，但如果你需要指定一个开始值100，那我们可以通过以下语句来实现：

```
mysql> CREATE TABLE insect
    -> (
    -> id INT UNSIGNED NOT NULL AUTO_INCREMENT = 100,
    -> PRIMARY KEY (id),
    -> name VARCHAR(30) NOT NULL, # type of insect
    -> date DATE NOT NULL, # date collected
    -> origin VARCHAR(30) NOT NULL # where collected
);
```

或者你也可以在表创建成功后，通过以下语句来实现：

```
mysql> ALTER TABLE t AUTO_INCREMENT = 100;
```

MySQL 处理重复数据

有些 MySQL 数据表中可能存在重复的记录，有些情况我们允许重复数据的存在，但有时候我们也需要删除这些重复的数据。

本章节我们将为大家介绍如何防止数据表出现重复数据及如何删除数据表中的重复数据。

防止表中出现重复数据

你可以在MySQL数据表中设置指定的字段为 **PRIMARY KEY**（主键） 或者 **UNIQUE**（唯一） 索引来保证数据的唯一性。

让我们尝试一个实例：下表中无索引及主键，所以该表允许出现多条重复记录。

```
CREATE TABLE person_tbl
(
    first_name CHAR(20),
    last_name CHAR(20),
    sex CHAR(10)
);
```

如果你想设置表中字段first_name，last_name数据不能重复，你可以设置双主键模式来设置数据的唯一性， 如果你设置了双主键，那么那个键的默认值不能为NULL，可设置为NOT NULL。如下所示：

```
CREATE TABLE person_tbl
(
   first_name CHAR(20) NOT NULL,
   last_name CHAR(20) NOT NULL,
   sex CHAR(10),
   PRIMARY KEY (last_name, first_name)
);
```

如果我们设置了唯一索引，那么在插入重复数据时，SQL语句将无法执行成功,并抛出错。

INSERT IGNORE INTO与INSERT INTO的区别就是INSERT IGNORE会忽略数据库中已经存在的数据，如果数据库没有数据，就插入新的数据，如果有数据的话就跳过这条数据。这样就可以保留数据库中已经存在数据，达到在间隙中插入数据的目的。

以下实例使用了INSERT IGNORE INTO，执行后不会出错，也不会向数据表中插入重复数据：

```
mysql> INSERT IGNORE INTO person_tbl (last_name, first_name)
    -> VALUES( 'Jay', 'Thomas');
Query OK, 1 row affected (0.00 sec)
mysql> INSERT IGNORE INTO person_tbl (last_name, first_name)
    -> VALUES( 'Jay', 'Thomas');
Query OK, 0 rows affected (0.00 sec)
```

INSERT IGNORE INTO当插入数据时，在设置了记录的唯一性后，如果插入重复数据，将不返回错误，只以警告形式返回。 而REPLACE INTO into如果存在primary 或 unique相同的记录，则先删除掉。再插入新记录。

另一种设置数据的唯一性方法是添加一个UNIQUE索引，如下所示：

```
CREATE TABLE person_tbl
(
   first_name CHAR(20) NOT NULL,
   last_name CHAR(20) NOT NULL,
   sex CHAR(10)
   UNIQUE (last_name, first_name)
);
```

统计重复数据

以下我们将统计表中 first_name 和 last_name的重复记录数：

```
mysql> SELECT COUNT(*) as repetitions, last_name, first_name
    -> FROM person_tbl
    -> GROUP BY last_name, first_name
    -> HAVING repetitions > 1;
```

以上查询语句将返回 person_tbl 表中重复的记录数。 一般情况下，查询重复的值，请执行以下操作：

- 确定哪一列包含的值可能会重复。
- 在列选择列表使用COUNT(*)列出的那些列。
- 在GROUP BY子句中列出的列。
- HAVING子句设置重复数大于1。

过滤重复数据

如果你需要读取不重复的数据可以在 SELECT 语句中使用 DISTINCT 关键字来过滤重复数据。

```
mysql> SELECT DISTINCT last_name, first_name
    -> FROM person_tbl
    -> ORDER BY last_name;
```

你也可以使用 GROUP BY 来读取数据表中不重复的数据：

```
mysql> SELECT last_name, first_name
    -> FROM person_tbl
    -> GROUP BY (last_name, first_name);
```

删除重复数据

如果你想删除数据表中的重复数据，你可以使用以下的SQL语句：

```
mysql> CREATE TABLE tmp SELECT last_name, first_name, sex
    ->                  FROM person_tbl;
    ->                  GROUP BY (last_name, first_name);
mysql> DROP TABLE person_tbl;
mysql> ALTER TABLE tmp RENAME TO person_tbl;
```

当然你也可以在数据表中添加 INDEX（索引） 和 PRIMAY KEY（主键）这种简单的方法来删除表中的重复记录。方法如下：

```
mysql> ALTER IGNORE TABLE person_tbl
    -> ADD PRIMARY KEY (last_name, first_name);
```

MySQL 及 SQL 注入

如果您通过网页获取用户输入的数据并将其插入一个MySQL数据库，那么就有可能发生SQL注入安全的问题。

本章节将为大家介绍如何防止SQL注入，并通过脚本来过滤SQL中注入的字符。

所谓SQL注入，就是通过把SQL命令插入到Web表单递交或输入域名或页面请求的查询字符串，最终达到欺骗服务器执行恶意的SQL命令。

我们永远不要信任用户的输入，我们必须认定用户输入的数据都是不安全的，我们都需要对用户输入的数据进行过滤处理。

以下实例中，输入的用户名必须为字母、数字及下划线的组合，且用户名长度为 8 到 20 个字符之间：

```
if (preg_match("/^\w{8,20}$/", $_GET['username'], $matches))
{
   $result = mysql_query("SELECT * FROM users
                          WHERE username=$matches[0]");
}
 else
{
   echo "username 输入异常";
}
```

让我们看下在没有过滤特殊字符时，出现的SQL情况：

```
// 设定$name 中插入了我们不需要的SQL语句
$name = "Qadir'; DELETE FROM users;";
mysql_query("SELECT * FROM users WHERE name='{$name}'");
```

以上的注入语句中，我们没有对 $name 的变量进行过滤，$name 中插入了我们不需要的SQL语句，将删除 users 表中的所有数据。

在PHP中的 mysql_query() 是不允许执行多个SQL语句的，但是在 SQLite 和 PostgreSQL 是可以同时执行多条SQL语句的，所以我们对这些用户的数据需要进行严格的验证。

防止SQL注入，我们需要注意以下几个要点：

- 1.永远不要信任用户的输入。对用户的输入进行校验，可以通过正则表达式，或限制长度；对单引号和 双"-"进行转换等。
- 2.永远不要使用动态拼装sql，可以使用参数化的sql或者直接使用存储过程进行数据查询存取。
- 3.永远不要使用管理员权限的数据库连接，为每个应用使用单独的权限有限的

数据库连接。
- v 4.不要把机密信息直接存放，加密或者hash掉密码和敏感的信息。
- 5.应用的异常信息应该给出尽可能少的提示，最好使用自定义的错误信息对原始错误信息进行包装
- 6.sql注入的检测方法一般采取辅助软件或网站平台来检测，软件一般采用sql注入检测工具jsky，网站平台就有亿思网站安全平台检测工具。MDCSOFT SCAN等。采用MDCSOFT-IPS可以有效的防御SQL注入，XSS攻击等。

防止**SQL**注入

在脚本语言，如Perl和PHP你可以对用户输入的数据进行转义从而来防止SQL注入。

PHP的MySQL扩展提供了mysql_real_escape_string()函数来转义特殊的输入字符。

```
if (get_magic_quotes_gpc())
{
  $name = stripslashes($name);
}
$name = mysql_real_escape_string($name);
mysql_query("SELECT * FROM users WHERE name='{$name}'");
```

Like语句中的注入

like查询时，如果用户输入的值有"*"*和*"%"*，则会出现这种情况：用户本来只是想查询*"abcd*"，查询结果中却有"abcd_"、"abcde"、"abcdf"等等；用户要查询"30%"（注：百分之三十）时也会出现问题。

在PHP脚本中我们可以使用addcslashes()函数来处理以上情况，如下实例：

```
$sub = addcslashes(mysql_real_escape_string("%something_"), "%_");
// $sub == \%something\_
mysql_query("SELECT * FROM messages WHERE subject LIKE '{$sub}%'");
```

addcslashes() 函数在指定的字符前添加反斜杠。

语法格式:

```
addcslashes(string,characters)
```

参数	描述
string	必需。规定要检查的字符串。
characters	可选。规定受 addcslashes() 影响的字符或字符范围。

具体应用可以查看：PHP addcslashes() 函数

MySQL 导出数据

MySQL中你可以使用**SELECT...INTO OUTFILE**语句来简单的导出数据到文本文件上。

使用 SELECT ... INTO OUTFILE 语句导出数据

以下实例中我们将数据表 tutorials_tbl 数据导出到 /tmp/tutorials.txt 文件中:

```
mysql> SELECT * FROM tutorials_tbl
    -> INTO OUTFILE '/tmp/tutorials.txt';
```

你可以通过命令选项来设置数据输出的指定格式，以下实例为导出 CSV 格式：

```
mysql> SELECT * FROM passwd INTO OUTFILE '/tmp/tutorials.txt'
    -> FIELDS TERMINATED BY ',' ENCLOSED BY '"'
    -> LINES TERMINATED BY '\r\n';
```

在下面的例子中，生成一个文件，各值用逗号隔开。这种格式可以被许多程序使用。

```
SELECT a,b,a+b INTO OUTFILE '/tmp/result.text'
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
LINES TERMINATED BY '\n'
FROM test_table;
```

SELECT ... INTO OUTFILE 语句有以下属性:

- LOAD DATA INFILE是SELECT ... INTO OUTFILE的逆操作，SELECT句法。为了将一个数据库的数据写入一个文件，使用SELECT ... INTO OUTFILE，为了将文件读回数据库，使用LOAD DATA INFILE。
- SELECT...INTO OUTFILE 'file_name'形式的SELECT可以把被选择的行写入一个文件中。该文件被创建到服务器主机上，因此您必须拥有FILE权限，才能使用此语法。
- 输出不能是一个已存在的文件。防止文件数据被篡改。
- 你需要有一个登陆服务器的账号来检索文件。否则 SELECT ... INTO OUTFILE 不会起任何作用。
- 在UNIX中，该文件被创建后是可读的，权限由MySQL服务器所拥有。这意味着，虽然你就可以读取该文件，但可能无法将其删除。

导出表作为原始数据

mysqldump是mysql用于转存储数据库的实用程序。它主要产生一个SQL脚本，其中包含从头重新创建数据库所必需的命令CREATE TABLE INSERT等。

使用mysqldump导出数据需要使用 --tab 选项来指定导出文件指定的目录，该目标必须是可写的。

以下实例将数据表 tutorials_tbl 导出到 /tmp 目录中：

```
$ mysqldump -u root -p --no-create-info \
            --tab=/tmp TUTORIALS tutorials_tbl
password ******
```

导出SQL格式的数据

导出SQL格式的数据到指定文件，如下所示：

```
$ mysqldump -u root -p TUTORIALS tutorials_tbl > dump.txt
password ******
```

以上命令创建的文件内容如下：

```
-- MySQL dump 8.23
--
-- Host: localhost    Database: TUTORIALS
---------------------------------------------------------
-- Server version       3.23.58

--
-- Table structure for table `tutorials_tbl`
--

CREATE TABLE tutorials_tbl (
  tutorial_id int(11) NOT NULL auto_increment,
  tutorial_title varchar(100) NOT NULL default '',
  tutorial_author varchar(40) NOT NULL default '',
  submission_date date default NULL,
  PRIMARY KEY  (tutorial_id),
  UNIQUE KEY AUTHOR_INDEX (tutorial_author)
) TYPE=MyISAM;

--
-- Dumping data for table `tutorials_tbl`
--

INSERT INTO tutorials_tbl
       VALUES (1,'Learn PHP','John Poul','2007-05-24');
INSERT INTO tutorials_tbl
       VALUES (2,'Learn MySQL','Abdul S','2007-05-24');
INSERT INTO tutorials_tbl
       VALUES (3,'JAVA Tutorial','Sanjay','2007-05-06');
```

如果你需要导出整个数据库的数据，可以使用以下命令：

```
$ mysqldump -u root -p TUTORIALS > database_dump.txt
password ******
```

如果需要备份所有数据库，可以使用以下命令：

```
$ mysqldump -u root -p --all-databases > database_dump.txt
password ******
```

--all-databases 选项在 MySQL 3.23.12 及以后版本加入。

该方法可用于实现数据库的备份策略。

将数据表及数据库拷贝至其他主机

如果你需要将数据拷贝至其他的 MySQL 服务器上, 你可以在 mysqldump 命令中指定数据库名及数据表。

在源主机上执行以下命令，将数据备份至 dump.txt 文件中:

```
$ mysqldump -u root -p database_name table_name > dump.txt
password *****
```

如果完整备份数据库，则无需使用特定的表名称。

如果你需要将备份的数据库导入到MySQL服务器中，可以使用以下命令，使用以下命令你需要确认数据库已经创建：

```
$ mysql -u root -p database_name < dump.txt
password *****
```

```
你也可以使用以下命令将导出的数据直接导入到远程的服务器上，但请确保两台服务器是相
$ mysqldump -u root -p database_name \
       | mysql -h other-host.com database_name
```

以上命令中使用了管道来将导出的数据导入到指定的远程主机上。

MySQL 导入数据

MySQL中可以使用两种简单的方式来导入MySQL导出的数据。

使用 LOAD DATA 导入数据

MySQL 中提供了LOAD DATA INFILE语句来插入数据。 以下实例中将从当前目录中读取文件 dump.txt ，将该文件中的数据插入到当前数据库的 mytbl 表中。

```
mysql> LOAD DATA LOCAL INFILE 'dump.txt' INTO TABLE mytbl;
```

如果指定LOCAL关键词，则表明从客户主机上按路径读取文件。如果没有指定，则文件在服务器上按路径读取文件。

你能明确地在LOAD DATA语句中指出列值的分隔符和行尾标记，但是默认标记是定位符和换行符。

两个命令的 FIELDS 和 LINES 子句的语法是一样的。两个子句都是可选的，但是如果两个同时被指定，FIELDS 子句必须出现在 LINES 子句之前。

如果用户指定一个 FIELDS 子句，它的子句 （TERMINATED BY、[OPTIONALLY] ENCLOSED BY 和 ESCAPED BY) 也是可选的，不过，用户必须至少指定它们中的一个。

```
mysql> LOAD DATA LOCAL INFILE 'dump.txt' INTO TABLE mytbl
  -> FIELDS TERMINATED BY ':'
  -> LINES TERMINATED BY '\r\n';
```

LOAD DATA 默认情况下是按照数据文件中列的顺序插入数据的，如果数据文件中的列与插入表中的列不一致，则需要指定列的顺序。

如，在数据文件中的列顺序是 a,b,c，但在插入表的列顺序为b,c,a，则数据导入语法如下：

```
mysql> LOAD DATA LOCAL INFILE 'dump.txt'
    -> INTO TABLE mytbl (b, c, a);
```

使用 mysqlimport 导入数据

mysqlimport客户端提供了LOAD DATA INFILEQL语句的一个命令行接口。mysqlimport的大多数选项直接对应LOAD DATA INFILE子句。

从文件 dump.txt 中将数据导入到 mytbl 数据表中, 可以使用以下命令：

```
$ mysqlimport -u root -p --local database_name dump.txt
password *****
```

mysqlimport命令可以指定选项来设置指定格式,命令语句格式如下：

```
$ mysqlimport -u root -p --local --fields-terminated-by=":" \
   --lines-terminated-by="\r\n"  database_name dump.txt
password *****
```

mysqlimport 语句中使用 --columns 选项来设置列的顺序：

```
$ mysqlimport -u root -p --local --columns=b,c,a \
    database_name dump.txt
password *****
```

mysqlimport的常用选项介绍

选项	功能
-d or --delete	新数据导入数据表中之前删除数据数据表中的所有信息
-f or --force	不管是否遇到错误，mysqlimport将强制继续插入数据
-i or --ignore	mysqlimport跳过或者忽略那些有相同唯一 关键字的行， 导入文件中的数据将被忽略。
-l or -lock-tables	数据被插入之前锁住表，这样就防止了， 你在更新数据库时，用户的查询和更新受到影响。
-r or -replace	这个选项与－i选项的作用相反；此选项将替代 表中有相同唯一关键字的记录。
--fields-enclosed-by= char	指定文本文件中数据的记录时以什么括起的， 很多情况下 数据以双引号括起。 默认的情况下数据是没有被字符括起的。
--fields-terminated-by=char	指定各个数据的值之间的分隔符，在句号分隔的文件中， 分隔符是句号。您可以用此选项指定数据之间的分隔符。 默认的分隔符是跳格符（Tab）
--lines-terminated-by=str	此选项指定文本文件中行与行之间数据的分隔字符串 或者字符。 默认的情况下mysqlimport以newline为行分隔符。 您可以选择用一个字符串来替代一个单个的字符： 一个新行或者一个回车。

mysqlimport命令常用的选项还有-v 显示版本（version）， -p 提示输入密码（password）等。

W3School SQLite教程

来源：SQLite教程

整理：飞龙

SQLite 基础

SQLite 简介

本教程帮助您了解什么是 SQLite，它与 SQL 之间的不同，为什么需要它，以及它的应用程序数据库处理方式。

SQLite是一个软件库，实现了自给自足的、无服务器的、零配置的、事务性的 SQL 数据库引擎。SQLite是一个增长最快的数据库引擎，这是在普及方面的增长，与它的尺寸大小无关。SQLite 源代码不受版权限制。

什么是 SQLite？

SQLite是一个进程内的库，实现了自给自足的、无服务器的、零配置的、事务性的 SQL 数据库引擎。它是一个零配置的数据库，这意味着与其他数据库一样，您不需要在系统中配置。

就像其他数据库，SQLite 引擎不是一个独立的进程，可以按应用程序需求进行静态或动态连接。SQLite 直接访问其存储文件。

为什么要用 SQLite？

- 不需要一个单独的服务器进程或操作的系统（无服务器的）。
- SQLite 不需要配置，这意味着不需要安装或管理。
- 一个完整的 SQLite 数据库是存储在一个单一的跨平台的磁盘文件。
- SQLite 是非常小的，是轻量级的，完全配置时小于 400KiB，省略可选功能配置时小于250KiB。
- SQLite 是自给自足的，这意味着不需要任何外部的依赖。
- SQLite 事务是完全兼容 ACID 的，允许从多个进程或线程安全访问。
- SQLite 支持 SQL92（SQL2）标准的大多数查询语言的功能。
- SQLite 使用 ANSI-C 编写的，并提供了简单和易于使用的 API。
- SQLite 可在 UNIX（Linux, Mac OS-X, Android, iOS）和 Windows（Win32, WinCE, WinRT）中运行。

历史

1. 2000 -- D. Richard Hipp 设计 SQLite 是为了不需要管理即可操作程序。
2. 2000 -- 在八月，SQLite1.0 发布 GNU 数据库管理器（GNU Database Manager）。

3. 2011 -- Hipp 宣布，向 SQLite DB 添加 UNQl 接口，开发 UNQLite（面向文档的数据库）。

SQLite 局限性

在 SQLite 中，SQL92 不支持的特性如下所示：

特性	描述
RIGHT OUTER JOIN	只实现了 LEFT OUTER JOIN。
FULL OUTER JOIN	只实现了 LEFT OUTER JOIN。
ALTER TABLE	支持 RENAME TABLE 和 ALTER TABLE 的 ADD COLUMN variants 命令，不支持 DROP COLUMN、ALTER COLUMN、ADD CONSTRAINT。
Trigger 支持	支持 FOR EACH ROW 触发器，但不支持 FOR EACH STATEMENT 触发器。
VIEWs	在 SQLite 中，视图是只读的。您不可以在视图上执行 DELETE、INSERT 或 UPDATE 语句。
GRANT 和 REVOKE	可以应用的唯一的访问权限是底层操作系统的正常文件访问权限。

SQLite 命令

与关系数据库进行交互的标准 SQLite 命令类似于 SQL。命令包括 CREATE、SELECT、INSERT、UPDATE、DELETE 和 DROP。这些命令基于它们的操作性质可分为以下几种：

DDL - 数据定义语言

命令	描述
CREATE	创建一个新的表，一个表的视图，或者数据库中的其他对象。
ALTER	修改数据库中的某个已有的数据库对象，比如一个表。
DROP	删除整个表，或者表的视图，或者数据库中的其他对象。

DML - 数据操作语言

命令	描述
INSERT	创建一条记录。
UPDATE	修改记录。
DELETE	删除记录。

DQL - 数据查询语言

命令	描述
SELECT	从一个或多个表中检索某些记录。

SQLite 安装

SQLite 的一个重要的特性是零配置的，这意味着不需要复杂的安装或管理。本章将讲解 Windows、Linux 和 Mac OS X 上的安装设置。

在 Windows 上安装 SQLite

- 请访问 SQLite 下载页面，从 Windows 区下载预编译的二进制文件。
- 您需要下载 **sqlite-shell-win32-*.zip** 和 **sqlite-dll-win32-*.zip** 压缩文件。
- 创建文件夹 C:\>sqlite，并在此文件夹下解压上面两个压缩文件，将得到 sqlite3.def、sqlite3.dll 和 sqlite3.exe 文件。
- 添加 C:\>sqlite 到 PATH 环境变量，最后在命令提示符下，使用 **sqlite3** 命令，将显示如下结果。

```
C:\>sqlite3
SQLite version 3.7.15.2 2013-01-09 11:53:05
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

在 Linux 上安装 SQLite

目前，几乎所有版本的 Linux 操作系统都附带 SQLite。所以，只要使用下面的命令来检查您的机器上是否已经安装了 SQLite。

```
$sqlite3
SQLite version 3.7.15.2 2013-01-09 11:53:05
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

如果没有看到上面的结果，那么就意味着没有在 Linux 机器上安装 SQLite。因此，让我们按照下面的步骤安装 SQLite：

- 请访问 SQLite 下载页面，从源代码区下载 **sqlite-autoconf-*.tar.gz**。
- 步骤如下：

```
$tar xvfz sqlite-autoconf-3071502.tar.gz
$cd sqlite-autoconf-3071502
$./configure --prefix=/usr/local
$make
$make install
```

上述步骤将在 Linux 机器上安装 SQLite，您可以按照上述讲解的进行验证。

在 Mac OS X 上安装 SQLite

最新版本的 Mac OS X 会预安装 SQLite，但是如果没有可用的安装，只需按照如下步骤进行：

- 请访问 SQLite 下载页面，从源代码区下载 **sqlite-autoconf-*.tar.gz**。
- 步骤如下：

```
$tar xvfz sqlite-autoconf-3071502.tar.gz
$cd sqlite-autoconf-3071502
$./configure --prefix=/usr/local
$make
$make install
```

上述步骤将在 Mac OS X 机器上安装 SQLite，您可以使用下列命令进行验证：

```
$sqlite3
SQLite version 3.7.15.2 2013-01-09 11:53:05
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

最后，在 SQLite 命令提示符下，使用 SQLite 命令做练习。

SQLite 命令

本章将向您讲解 SQLite 编程人员所使用的简单却有用的命令。些命令被称为 SQLite 的点命令，这些命令的不同之处在于它们不以分号（;）结束。

让我们在命令提示符下键入一个简单的 **sqlite3** 命令，在 SQLite 命令提示符下，您可以使用各种 SQLite 命令。

```
$sqlite3
SQLite version 3.3.6
Enter ".help" for instructions
sqlite>
```

如需获取可用的点命令的清单，可以在任何时候输入 ".help"。例如：

```
sqlite>.help
```

上面的命令会显示各种重要的 SQLite 点命令的列表，如下所示：

命令	描述
.backup ?DB? FILE	备份 DB 数据库（默认是 "main"）到 FILE 文件。
.bail ON|OFF	发生错误后停止。默认为 OFF。
.databases	列出附加数据库的名称和文件。
.dump ?TABLE?	以 SQL 文本格式转储数据库。如果指定了 TABLE 表，则只转储匹配 LIKE 模式的 TABLE 表。
.echo ON|OFF	开启或关闭 echo 命令。
.exit	退出 SQLite 提示符。
.explain ON|OFF	开启或关闭适合于 EXPLAIN 的输出模式。如果没有带参数，则为 EXPLAIN on，及开启 EXPLAIN。
.header(s) ON|OFF	开启或关闭头部显示。
.help	显示消息。
.import FILE TABLE	导入来自 FILE 文件的数据到 TABLE 表中。

.indices ? TABLE?	显示所有索引的名称。如果指定了 TABLE 表，则只显示匹配 LIKE 模式的 TABLE 表的索引。
.load FILE ?ENTRY?	加载一个扩展库。
.log FILE|off	开启或关闭日志。FILE 文件可以是 stderr（标准错误）/stdout（标准输出）。
.mode MODE	设置输出模式，MODE 可以是下列之一： **csv** 逗号分隔的值 **column** 左对齐的列 **html** HTML 的 `<table>` 代码 **insert** TABLE 表的 SQL 插入（insert）语句 **line** 每行一个值 **list** 由 .separator 字符串分隔的值 **tabs** 由 Tab 分隔的值 **tcl** TCL 列表元素
.nullvalue STRING	在 NULL 值的地方输出 STRING 字符串。
.output FILENAME	发送输出到 FILENAME 文件。
.output stdout	发送输出到屏幕。
.print STRING...	逐字地输出 STRING 字符串。
.prompt MAIN CONTINUE	替换标准提示符。
.quit	退出 SQLite 提示符。
.read FILENAME	执行 FILENAME 文件中的 SQL。
.schema ? TABLE?	显示 CREATE 语句。如果指定了 TABLE 表，则只显示匹配 LIKE 模式的 TABLE 表。
.separator STRING	改变输出模式和 .import 所使用的分隔符。
.show	显示各种设置的当前值。
.stats ON|OFF	开启或关闭统计。
.tables ? PATTERN?	列出匹配 LIKE 模式的表的名称。
.timeout MS	尝试打开锁定的表 MS 微秒。
.width NUM	

NUM	
.timer ON|OFF	开启或关闭 CPU 定时器测量。

让我们尝试使用 **.show** 命令，来查看 SQLite 命令提示符的默认设置。

```
sqlite>.show
     echo: off
  explain: off
  headers: off
     mode: column
nullvalue: ""
   output: stdout
separator: "|"
    width:
sqlite>
```

> 确保 sqlite> 提示符与点命令之间没有空格，否则将无法正常工作。

格式化输出

您可以使用下列的点命令来格式化输出为本教程下面所列出的格式：

```
sqlite>.header on
sqlite>.mode column
sqlite>.timer on
sqlite>
```

上面设置将产生如下格式的输出：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
CPU Time: user 0.000000 sys 0.000000
```

sqlite_master 表格

主表中保存数据库表的关键信息，并把它命名为 **sqlite_master**。如要查看表概要，可按如下操作：

```
sqlite>.schema sqlite_master
```

这将产生如下结果：

```
CREATE TABLE sqlite_master (
  type text,
  name text,
  tbl_name text,
  rootpage integer,
  sql text
);
```

SQLite 语法

SQLite 是遵循一套独特的称为语法的规则和准则。本教程列出了所有基本的 SQLite 语法，向您提供了一个 SQLite 快速入门。

大小写敏感性

有个重要的点值得注意，SQLite 是不区分大小写的，但也有一些命令是大小写敏感的，比如 **GLOB** 和 **glob** 在 SQLite 的语句中有不同的含义。

注释

SQLite 注释是附加的注释，可以在 SQLite 代码中添加注释以增加其可读性，他们可以出现在任何空白处，包括在表达式内和其他 SQL 语句的中间，但它们不能嵌套。

SQL 注释以两个连续的 "-" 字符（ASCII 0x2d）开始，并扩展至下一个换行符（ASCII 0x0a）或直到输入结束，以先到者为准。

您也可以使用 C 风格的注释，以 "/*" 开始，并扩展至下一个 "*/" 字符对或直到输入结束，以先到者为准。C 庚哥的注释可以跨越多行。

```
sqlite>.help -- This is a single line comment
```

SQLite 语句

所有的 SQLite 语句可以以任何关键字开始，如 SELECT、INSERT、UPDATE、DELETE、ALTER、DROP 等，所有的语句以分号（;）结束。

SQLite ANALYZE 语句：

```
ANALYZE;
or
ANALYZE database_name;
or
ANALYZE database_name.table_name;
```

SQLite AND/OR 子句：

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  CONDITION-1 {AND|OR} CONDITION-2;
```

SQLite ALTER TABLE 语句：

```
ALTER TABLE table_name ADD COLUMN column_def...;
```

SQLite ALTER TABLE 语句（Rename）：

```
ALTER TABLE table_name RENAME TO new_table_name;
```

SQLite ATTACH DATABASE 语句：

```
ATTACH DATABASE 'DatabaseName' As 'Alias-Name';
```

SQLite BEGIN TRANSACTION 语句：

```
BEGIN;
or
BEGIN EXCLUSIVE TRANSACTION;
```

SQLite BETWEEN 子句：

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  column_name BETWEEN val-1 AND val-2;
```

SQLite COMMIT 语句：

```
COMMIT;
```

SQLite CREATE INDEX 语句：

```
CREATE INDEX index_name
ON table_name ( column_name COLLATE NOCASE );
```

SQLite CREATE UNIQUE INDEX 语句：

```
CREATE UNIQUE INDEX index_name
ON table_name ( column1, column2,...columnN);
```

SQLite CREATE TABLE 语句：

```
CREATE TABLE table_name(
   column1 datatype,
   column2 datatype,
   column3 datatype,
   .....
   columnN datatype,
   PRIMARY KEY( one or more columns )
);
```

SQLite CREATE TRIGGER 语句：

```
CREATE TRIGGER database_name.trigger_name
BEFORE INSERT ON table_name FOR EACH ROW
BEGIN
   stmt1;
   stmt2;
   ....
END;
```

SQLite CREATE VIEW 语句：

```
CREATE VIEW database_name.view_name  AS
SELECT statement....;
```

SQLite CREATE VIRTUAL TABLE 语句：

```
CREATE VIRTUAL TABLE database_name.table_name USING weblog( access.
or
CREATE VIRTUAL TABLE database_name.table_name USING fts3( );
```

SQLite COMMIT TRANSACTION 语句：

```
COMMIT;
```

SQLite COUNT 子句：

```
SELECT COUNT(column_name)
FROM   table_name
WHERE  CONDITION;
```

SQLite DELETE 语句：

```
DELETE FROM table_name
WHERE  {CONDITION};
```

SQLite DETACH DATABASE 语句：

```
DETACH DATABASE 'Alias-Name';
```

SQLite DISTINCT 子句：

```
SELECT DISTINCT column1, column2....columnN
FROM   table_name;
```

SQLite DROP INDEX 语句：

```
DROP INDEX database_name.index_name;
```

SQLite DROP TABLE 语句：

```
DROP TABLE database_name.table_name;
```

SQLite DROP VIEW 语句：

```
DROP INDEX database_name.view_name;
```

SQLite DROP TRIGGER 语句：

```
DROP INDEX database_name.trigger_name;
```

SQLite EXISTS 子句：

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  column_name EXISTS (SELECT * FROM   table_name );
```

SQLite EXPLAIN 语句：

```
EXPLAIN INSERT statement...;
or
EXPLAIN QUERY PLAN SELECT statement...;
```

SQLite GLOB 子句：

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  column_name GLOB { PATTERN };
```

SQLite GROUP BY 子句：

```
SELECT SUM(column_name)
FROM   table_name
WHERE  CONDITION
GROUP BY column_name;
```

SQLite HAVING 子句：

```
SELECT SUM(column_name)
FROM   table_name
WHERE  CONDITION
GROUP BY column_name
HAVING (arithematic function condition);
```

SQLite INSERT INTO 语句：

```
INSERT INTO table_name( column1, column2....columnN)
VALUES ( value1, value2....valueN);
```

SQLite IN 子句：

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  column_name IN (val-1, val-2,...val-N);
```

SQLite Like 子句：

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  column_name LIKE { PATTERN };
```

SQLite NOT IN 子句：

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  column_name NOT IN (val-1, val-2,...val-N);
```

SQLite ORDER BY 子句：

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  CONDITION
ORDER BY column_name {ASC|DESC};
```

SQLite PRAGMA 语句：

```
PRAGMA pragma_name;

For example:

PRAGMA page_size;
PRAGMA cache_size = 1024;
PRAGMA table_info(table_name);
```

SQLite RELEASE SAVEPOINT 语句：

```
RELEASE savepoint_name;
```

SQLite REINDEX 语句：

```
REINDEX collation_name;
REINDEX database_name.index_name;
REINDEX database_name.table_name;
```

SQLite ROLLBACK 语句：

```
ROLLBACK;
or
ROLLBACK TO SAVEPOINT savepoint_name;
```

SQLite SAVEPOINT 语句：

```
SAVEPOINT savepoint_name;
```

SQLite SELECT 语句：

```
SELECT column1, column2....columnN
FROM   table_name;
```

SQLite UPDATE 语句：

```
UPDATE table_name
SET column1 = value1, column2 = value2....columnN=valueN
[ WHERE  CONDITION ];
```

SQLite VACUUM 语句：

```
VACUUM;
```

SQLite WHERE 子句：

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  CONDITION;
```

SQLite 数据类型

SQLite 数据类型是一个用来指定任何对象的数据类型的属性。SQLite 中的每一列，每个变量和表达式都有相关的数据类型。

您可以在创建表的同时使用这些数据类型。SQLite 使用一个更普遍的动态类型系统。在 SQLite 中，值的数据类型与值本身是相关的，而不是与它的容器相关。

SQLite 存储类

每个存储在 SQLite 数据库中的值都具有以下存储类之一：

存储类	描述
NULL	值是一个 NULL 值。
INTEGER	值是一个带符号的整数，根据值的大小存储在 1、2、3、4、6 或 8 字节中。
REAL	值是一个浮点值，存储为 8 字节的 IEEE 浮点数字。
TEXT	值是一个文本字符串，使用数据库编码（UTF-8、UTF-16BE 或 UTF-16LE）存储。
BLOB	值是一个 blob 数据，完全根据它的输入存储。

SQLite 的存储类稍微比数据类型更普遍。INTEGER 存储类，例如，包含 6 种不同的不同长度的整数数据类型。

SQLite Affinity 类型

SQLite 支持列上的类型 *affinity* 概念。任何列仍然可以存储任何类型的数据，但列的首选存储类是它的 **affinity**。在 SQLite3 数据库中，每个表的列分配为以下类型的 affinity 之一：

Affinity	描述
TEXT	该列使用存储类 NULL、TEXT 或 BLOB 存储所有数据。
NUMERIC	该列可以包含使用所有五个存储类的值。
INTEGER	与带有 NUMERIC affinity 的列相同，在 CAST 表达式中带有异常。
REAL	与带有 NUMERIC affinity 的列相似，不同的是，它会强制把整数值转换为浮点表示。
NONE	带有 affinity NONE 的列，不会优先使用哪个存储类，也不会尝试把数据从一个存储类强制转换为另一个存储类。

SQLite Affinity 及类型名称

下表列出了当创建 SQLite3 表时可使用的各种数据类型名称，同时也显示了相应的应用 Affinity：

数据类型	Affinity
INT INTEGER TINYINT SMALLINT MEDIUMINT BIGINT UNSIGNED BIG INT INT2 INT8	INTEGER
CHARACTER(20) VARCHAR(255) VARYING CHARACTER(255) NCHAR(55) NATIVE CHARACTER(70) NVARCHAR(100) TEXT CLOB	TEXT
BLOB "no datatype specified"	NONE
REAL DOUBLE DOUBLE PRECISION FLOAT	REAL
NUMERIC DECIMAL(10,5) BOOLEAN DATE DATETIME	NUMERIC

Boolean 数据类型

SQLite 没有单独的 Boolean 存储类。相反，布尔值被存储为整数 0（false）和 1（true）。

Date 与 Time 数据类型

SQLite 没有一个单独的用于存储日期和/或时间的存储类，但 SQLite 能够把日期和时间存储为 TEXT、REAL 或 INTEGER 值。

存储类	日期格式
TEXT	格式为 "YYYY-MM-DD HH:MM:SS.SSS" 的日期。
REAL	从公元前 4714 年 11 月 24 日格林尼治时间的正午开始算起的天数。
INTEGER	从 1970-01-01 00:00:00 UTC 算起的秒数。

您可以以任何上述格式来存储日期和时间，并且可以使用内置的日期和时间函数来自由转换不同格式。

SQLite 创建数据库

SQLite 的 **sqlite3** 命令被用来创建新的 SQLite 数据库。您不需要任何特殊的权限即可创建一个数据。

语法

sqlite3 命令的基本语法如下：

```
$sqlite3 DatabaseName.db
```

通常情况下，数据库名称在 RDBMS 内应该是唯一的。

实例

如果您想创建一个新的数据库 <testDB.db>，SQLITE3 语句如下所示：

```
$sqlite3 testDB.db
SQLite version 3.7.15.2 2013-01-09 11:53:05
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

上面的命令将在当前目录下创建一个文件 **testDB.db**。该文件将被 SQLite 引擎用作数据库。如果您已经注意到 sqlite3 命令在成功创建数据库文件之后，将提供一个 **sqlite>** 提示符。

一旦数据库被创建，您就可以使用 SQLite 的 **.databases** 命令来检查它是否在数据库列表中，如下所示：

```
sqlite>.databases
seq  name             file
---  ---------------  ----------------------
0    main             /home/sqlite/testDB.db
```

您可以使用 SQLite **.quit** 命令退出 sqlite 提示符，如下所示：

```
sqlite>.quit
$
```

.dump 命令

您可以在命令提示符中使用 SQLite **.dump** 点命令来导出完整的数据库在一个文本文件中，如下所示：

```
$sqlite3 testDB.db .dump > testDB.sql
```

上面的命令将转换整个 **testDB.db** 数据库的内容到 SQLite 的语句中，并将其转储到 ASCII 文本文件 **testDB.sql** 中。您可以通过简单的方式从生成的 testDB.sql 恢复，如下所示：

```
$sqlite3 testDB.db < testDB.sql
```

此时的数据库是空的，一旦数据库中有表和数据，您可以尝试上述两个程序。现在，让我们继续学习下一章。

SQLite 附加数据库

假设这样一种情况，当在同一时间有多个数据库可用，您想使用其中的任何一个。SQLite 的 **ATTACH DTABASE** 语句是用来选择一个特定的数据库，使用该命令后，所有的 SQLite 语句将在附加的数据库下执行。

语法

SQLite 的 ATTACH DATABASE 语句的基本语法如下：

```
ATTACH DATABASE 'DatabaseName' As 'Alias-Name';
```

如果数据库尚未被创建，上面的命令将创建一个数据库，如果数据库已存在，则把数据库文件名称与逻辑数据库 'Alias-Name' 绑定在一起。

实例

如果想附加一个现有的数据库 **testDB.db**，则 ATTACH DATABASE 语句将如下所示：

```
sqlite> ATTACH DATABASE 'testDB.db' as 'TEST';
```

使用 SQLite **.database** 命令来显示附加的数据库。

```
sqlite> .database
seq  name             file
---  ---------------  ----------------------
0    main             /home/sqlite/testDB.db
2    test             /home/sqlite/testDB.db
```

数据库名称 **main** 和 **temp** 被保留用于主数据库和存储临时表及其他临时数据对象的数据库。这两个数据库名称可用于每个数据库连接，且不应该被用于附加，否则将得到一个警告消息，如下所示：

```
sqlite>  ATTACH DATABASE 'testDB.db' as 'TEMP';
Error: database TEMP is already in use
sqlite>  ATTACH DATABASE 'testDB.db' as 'main';
Error: database TEMP is already in use
```

SQLite 分离数据库

SQLite的 **DETACH DTABASE** 语句是用来把命名数据库从一个数据库连接分离和游离出来，连接是之前使用 ATTACH 语句附加的。如果同一个数据库文件已经被附加上多个别名，DETACH 命令将只断开给定名称的连接，而其余的仍然有效。您无法分离 **main** 或 **temp** 数据库。

> 如果数据库是在内存中或者是临时数据库，则该数据库将被摧毁，且内容将会丢失。

语法

SQLite 的 DETACH DATABASE 'Alias-Name' 语句的基本语法如下：

```
DETACH DATABASE 'Alias-Name';
```

在这里，'Alias-Name' 与您之前使用 ATTACH 语句附加数据库时所用到的别名相同。

实例

假设在前面的章节中您已经创建了一个数据库，并给它附加了 'test' 和 'currentDB'，使用 .database 命令，我们可以看到：

```
sqlite>.databases
seq  name             file
---  ---------------  ----------------------
0    main             /home/sqlite/testDB.db
2    test             /home/sqlite/testDB.db
3    currentDB        /home/sqlite/testDB.db
```

现在，让我们尝试把 'currentDB' 从 testDB.db 中分离出来，如下所示：

```
sqlite> DETACH DATABASE 'currentDB';
```

现在，如果检查当前附加的数据库，您会发现，testDB.db 仍与 'test' 和 'main' 保持连接。

```
sqlite>.databases
seq  name             file
---  ---------------  ----------------------
0    main             /home/sqlite/testDB.db
2    test             /home/sqlite/testDB.db
```

SQLite 创建表

SQLite 的 **CREATE TABLE** 语句用于在任何给定的数据库创建一个新表。创建基本表，涉及到命名表、定义列及每一列的数据类型。

语法

CREATE TABLE 语句的基本语法如下：

```
CREATE TABLE database_name.table_name(
   column1 datatype  PRIMARY KEY(one or more columns),
   column2 datatype,
   column3 datatype,
   .....
   columnN datatype,
);
```

CREATE TABLE 是告诉数据库系统创建一个新表的关键字。CREATE TABLE 语句后跟着表的唯一的名称或标识。您也可以选择指定带有 *table_name* 的 *database_name*。

实例

下面是一个实例，它创建了一个 COMPANY 表，ID 作为主键，NOT NULL 的约束表示在表中创建纪录时这些字段不能为 NULL：

```
sqlite> CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

让我们再创建一个表，我们将在随后章节的练习中使用：

```
sqlite> CREATE TABLE DEPARTMENT(
   ID INT PRIMARY KEY      NOT NULL,
   DEPT           CHAR(50) NOT NULL,
   EMP_ID         INT      NOT NULL
);
```

您可以使用 SQLIte 命令中的 **.tables** 命令来验证表是否已成功创建，该命令用于列出附加数据库中的所有表。

```
sqlite>.tables
COMPANY     DEPARTMENT
```

在这里，可以看到 COMPANY 表出现两次，一个是主数据库的 COMPANY 表，一个是为 testDB.db 创建的 'test' 别名的 test.COMPANY 表。您可以使用 SQLite **.schema** 命令得到表的完整信息，如下所示：

```
sqlite>.schema COMPANY
CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

SQLite 删除表

SQLite 的 **DROP TABLE** 语句用来删除表定义及其所有相关数据、索引、触发器、约束和该表的权限规范。

> 使用此命令时要特别注意，因为一旦一个表被删除，表中所有信息也将永远丢失。

语法

DROP TABLE 语句的基本语法如下。您可以选择指定带有表名的数据库名称，如下所示：

```
DROP TABLE database_name.table_name;
```

实例

让我们先确认 COMPANY 表已经存在，然后我们将其从数据库中删除。

```
sqlite>.tables
COMPANY       test.COMPANY
```

这意味着 COMPANY 表已存在数据库中，接下来让我们把它从数据库中删除，如下：

```
sqlite>DROP TABLE COMPANY;
sqlite>
```

现在，如果尝试 .TABLES 命令，那么将无法找到 COMPANY 表了：

```
sqlite>.tables
sqlite>
```

显示结果为空，意味着已经成功从数据库删除表。

SQLite Insert 语句

SQLite 的 **INSERT INTO** 语句用于向数据库的某个表中添加新的数据行。

语法

INSERT INTO 语句有两种基本语法，如下所示：

```
INSERT INTO TABLE_NAME (column1, column2, column3,...columnN)]
VALUES (value1, value2, value3,...valueN);
```

在这里，column1, column2,...columnN 是要插入数据的表中的列的名称。

如果要为表中的所有列添加值，您也可以不需要在 SQLite 查询中指定列名称。但要确保值的顺序与列在表中的顺序一致。SQLite 的 INSERT INTO 语法如下：

```
INSERT INTO TABLE_NAME VALUES (value1,value2,value3,...valueN);
```

实例

假设您已经在 testDB.db 中创建了 COMPANY表，如下所示：

```
sqlite> CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

现在，下面的语句将在 COMPANY 表中创建六个记录：

```
INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Paul', 32, 'California', 20000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (2, 'Allen', 25, 'Texas', 15000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (3, 'Teddy', 23, 'Norway', 20000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (5, 'David', 27, 'Texas', 85000.00 );

INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (6, 'Kim', 22, 'South-Hall', 45000.00 );
```

您也可以使用第二种语法在 COMPANY 表中创建一个记录，如下所示：

```
INSERT INTO COMPANY VALUES (7, 'James', 24, 'Houston', 10000.00 );
```

上面的所有语句将在 COMPANY 表中创建下列记录。下一章会教您如何从一个表中显示所有这些记录。

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

使用一个表来填充另一个表

您可以通过在一个有一组字段的表上使用 select 语句，填充数据到零一个表中。下面是语法：

```
INSERT INTO first_table_name [(column1, column2, ... columnN)]
   SELECT column1, column2, ...columnN
   FROM second_table_name
   [WHERE condition];
```

您暂时可以先跳过上面的语句，可以先学习后面章节中介绍的 SELECT 和 WHERE 子句。

SQLite Select 语句

SQLite 的 **SELECT** 语句用于从 SQLite 数据库表中获取数据，以结果表的形式返回数据。这些结果表也被称为结果集。

语法

SQLite 的 SELECT 语句的基本语法如下：

```
SELECT column1, column2, columnN FROM table_name;
```

在这里，column1, column2...是表的字段，他们的值即是您要获取的。如果您想获取所有可用的字段，那么可以使用下面的语法：

```
SELECT * FROM table_name;
```

实例

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面是一个实例，使用 SELECT 语句获取并显示所有这些记录。在这里，前三个命令被用来设置正确格式化的输出。

```
sqlite>.header on
sqlite>.mode column
sqlite> SELECT * FROM COMPANY;
```

最后，将得到以下的结果：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

如果只想获取 COMPANY 表中指定的字段，则使用下面的查询：

```
sqlite> SELECT ID, NAME, SALARY FROM COMPANY;
```

上面的查询会产生以下结果：

```
ID          NAME        SALARY
----------  ----------  ----------
1           Paul        20000.0
2           Allen       15000.0
3           Teddy       20000.0
4           Mark        65000.0
5           David       85000.0
6           Kim         45000.0
7           James       10000.0
```

设置输出列的宽度

有时，由于要显示的列的默认宽度导致 **.mode column**，这种情况下，输出被截断。此时，您可以使用 **.width num, num....** 命令设置显示列的宽度，如下所示：

```
sqlite>.width 10, 20, 10
sqlite>SELECT * FROM COMPANY;
```

上面的 **.width** 命令设置第一列的宽度为 10，第二列的宽度为 20，第三列的宽度为 10。因此上述 SELECT 语句将得到以下结果：

```
ID          NAME                  AGE         ADDRESS     SALARY
----------  --------------------  ----------  ----------  ---------
1           Paul                  32          California  20000.0
2           Allen                 25          Texas       15000.0
3           Teddy                 23          Norway      20000.0
4           Mark                  25          Rich-Mond   65000.0
5           David                 27          Texas       85000.0
6           Kim                   22          South-Hall  45000.0
7           James                 24          Houston     10000.0
```

Schema 信息

因为所有的点命令只在 SQLite 提示符中可用，所以当您进行带有 SQLite 的编程时，您要使用下面的带有 **sqlite_master** 表的 SELECT 语句来列出所有在数据库中创建的表：

```
sqlite> SELECT tbl_name FROM sqlite_master WHERE type = 'table';
```

假设在 testDB.db 中已经存在唯一的 COMPANY 表，则将产生以下结果：

```
tbl_name
----------
COMPANY
```

您可以列出关于 COMPANY 表的完整信息，如下所示：

```
sqlite> SELECT sql FROM sqlite_master WHERE type = 'table' AND tbl_
```

假设在 testDB.db 中已经存在唯一的 COMPANY 表，则将产生以下结果：

```
CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
)
```

SQLite 运算符

SQLite 运算符是什么？

运算符是一个保留字或字符，主要用于 SQLite 语句的 WHERE 子句中执行操作，如比较和算术运算。

运算符用于指定 SQLite 语句中的条件，并在语句中连接多个条件。

- 算术运算符
- 比较运算符
- 逻辑运算符
- 位运算符

SQLite 算术运算符

假设变量 a=10，变量 b=20，则：

运算符	描述	实例
+	加法 - 把运算符两边的值相加	a + b 将得到 30
-	减法 - 左操作数减去右操作数	a - b 将得到 -10
*	乘法 - 把运算符两边的值相乘	a * b 将得到 200
/	除法 - 左操作数除以右操作数	b / a 将得到 2
%	取模 - 左操作数除以右操作数后得到的余数	b % a will give 0

实例

下面是 SQLite 算术运算符的简单实例：

```
sqlite> .mode line
sqlite> select 10 + 20;
10 + 20 = 30

sqlite> select 10 - 20;
10 - 20 = -10

sqlite> select 10 * 20;
10 * 20 = 200

sqlite> select 10 / 5;
10 / 5 = 2

sqlite> select 12 %  5;
12 %  5 = 2
```

SQLite 比较运算符

假设变量 a=10，变量 b=20，则：

运算符	描述	实例
==	检查两个操作数的值是否相等，如果相等则条件为真。	(a == b) 不为真。
=	检查两个操作数的值是否相等，如果相等则条件为真。	(a = b) 不为真。
!=	检查两个操作数的值是否相等，如果不相等则条件为真。	(a != b) 为真。
<>	检查两个操作数的值是否相等，如果不相等则条件为真。	(a <> b) 为真。
>	检查左操作数的值是否大于右操作数的值，如果是则条件为真。	(a > b) 不为真。
<	检查左操作数的值是否小于右操作数的值，如果是则条件为真。	(a < b) 为真。
>=	检查左操作数的值是否大于等于右操作数的值，如果是则条件为真。	(a >= b) 不为真。
<=	检查左操作数的值是否小于等于右操作数的值，如果是则条件为真。	(a <= b) 为真。
!<	检查左操作数的值是否不小于右操作数的值，如果是则条件为真。	(a !< b) 为假。
!>	检查左操作数的值是否不大于右操作数的值，如果是则条件为真。	(a !> b) 为真。

实例

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面的实例演示了各种 SQLite 比较运算符的用法。

> 在这里，我们使用 **WHERE** 子句，这将会在后边单独的一个章节中讲解，但现在您需要明白，WHERE 子句是用来设置 SELECT 语句的条件语句。

下面的 SELECT 语句列出了 SALARY 大于 50,000.00 的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE SALARY > 50000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

下面的 SELECT 语句列出了 SALARY 等于 20,000.00 的所有记录：

```
sqlite>  SELECT * FROM COMPANY WHERE SALARY = 20000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
3           Teddy       23          Norway      20000.0
```

下面的 SELECT 语句列出了 SALARY 不等于 20,000.00 的所有记录：

```
sqlite>  SELECT * FROM COMPANY WHERE SALARY != 20000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面的 SELECT 语句列出了 SALARY 不等于 20,000.00 的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE SALARY <> 20000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面的 SELECT 语句列出了 SALARY 大于等于 65,000.00 的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE SALARY >= 65000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

SQLite 逻辑运算符

下面是 SQLite 中所有的逻辑运算符列表。

运算符	描述
AND	AND 运算符允许在一个 SQL 语句的 WHERE 子句中的多个条件的存在。
BETWEEN	BETWEEN 运算符用于在给定最小值和最大值范围内的一系列值中搜索值。
EXISTS	EXISTS 运算符用于在满足一定条件的指定表中搜索行的存在。
IN	IN 运算符用于把某个值与一系列指定列表的值进行比较。
NOT IN	IN 运算符的对立面，用于把某个值与不在一系列指定列表的值进行比较。
LIKE	LIKE 运算符用于把某个值与使用通配符运算符的相似值进行比较。
GLOB	GLOB 运算符用于把某个值与使用通配符运算符的相似值进行比较。GLOB 与 LIKE 不同之处在于，它是大小写敏感的。
NOT	NOT 运算符是所用的逻辑运算符的对立面。比如 NOT EXISTS、NOT BETWEEN、NOT IN，等等。它是否定运算符。
OR	OR 运算符用于结合一个 SQL 语句的 WHERE 子句中的多个条件。
IS NULL	NULL 运算符用于把某个值与 NULL 值进行比较。
IS	IS 运算符与 = 相似。
IS NOT	IS NOT 运算符与 != 相似。
||	连接两个不同的字符串，得到一个新的字符串。
UNIQUE	UNIQUE 运算符搜索指定表中的每一行，确保唯一性（无重复）。

实例

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面的实例演示了 SQLite 逻辑运算符的用法。

下面的 SELECT 语句列出了 AGE 大于等于 25 且工资大于等于 65000.00 的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE AGE >= 25 AND SALARY >= 65000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

下面的 SELECT 语句列出了 AGE 大于等于 25 或工资大于等于 65000.00 的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE AGE >= 25 OR SALARY >= 65000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

下面的 SELECT 语句列出了 AGE 不为 NULL 的所有记录，结果显示所有的记录，意味着没有一个记录的 AGE 等于 NULL：

```
sqlite>  SELECT * FROM COMPANY WHERE AGE IS NOT NULL;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面的 SELECT 语句列出了 NAME 以 'Ki' 开始的所有记录，'Ki' 之后的字符不做限制：

```
sqlite> SELECT * FROM COMPANY WHERE NAME LIKE 'Ki%';
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
6           Kim         22          South-Hall  45000.0
```

下面的 SELECT 语句列出了 NAME 以 'Ki' 开始的所有记录，'Ki' 之后的字符不做限制：

```
sqlite> SELECT * FROM COMPANY WHERE NAME GLOB 'Ki*';
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
6           Kim         22          South-Hall  45000.0
```

下面的 SELECT 语句列出了 AGE 的值为 25 或 27 的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE AGE IN ( 25, 27 );
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

下面的 SELECT 语句列出了 AGE 的值既不是 25 也不是 27 的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE AGE NOT IN ( 25, 27 );
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
3           Teddy       23          Norway      20000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面的 SELECT 语句列出了 AGE 的值在 25 与 27 之间的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE AGE BETWEEN 25 AND 27;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

下面的 SELECT 语句使用 SQL 子查询，子查询查找 SALARY > 65000 的带有 AGE 字段的所有记录，后边的 WHERE 子句与 EXISTS 运算符一起使用，列出了外查询中的 AGE 存在于子查询返回的结果中的所有记录：

```
sqlite> SELECT AGE FROM COMPANY
        WHERE EXISTS (SELECT AGE FROM COMPANY WHERE SALARY > 65000)
AGE
----------
32
25
23
25
27
22
24
```

下面的 SELECT 语句使用 SQL 子查询，子查询查找 SALARY > 65000 的带有 AGE 字段的所有记录，后边的 WHERE 子句与 > 运算符一起使用，列出了外查询中的 AGE 大于子查询返回的结果中的年龄的所有记录：

```
sqlite> SELECT * FROM COMPANY
        WHERE AGE > (SELECT AGE FROM COMPANY WHERE SALARY > 65000);
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
```

SQLite 位运算符

位运算符作用于位，并逐位执行操作。真值表 & 和 | 如下：

p	q	p & q	p | q
0	0	0	0
0	1	0	1
1	1	1	1
1	0	0	1

假设如果 A = 60，且 B = 13，现在以二进制格式，它们如下所示：

```
A = 0011 1100

B = 0000 1101

-----------------

A&B = 0000 1100

A|B = 0011 1101

~A  = 1100 0011
```

下表中列出了 SQLite 语言支持的位运算符。假设变量 A=60，变量 B=13，则：

运算符	描述	实例
&	如果同时存在于两个操作数中，二进制 AND 运算符复制一位到结果中。	(A & B) 将得到 12，即为 0000 1100
|	如果存在于任一操作数中，二进制 OR 运算符复制一位到结果中。	(A | B) 将得到 61，即为 0011 1101
~	二进制补码运算符是一元运算符，具有"翻转"位效应。	(~A) 将得到 -61，即为 1100 0011，2 的补码形式，带符号的二进制数。
<<	二进制左移运算符。左操作数的值向左移动右操作数指定的位数。	A << 2 将得到 240，即为 1111 0000
>>	二进制右移运算符。左操作数的值向右移动右操作数指定的位数。	A >> 2 将得到 15，即为 0000 1111

实例

下面的实例演示了 SQLite 位运算符的用法：

```
sqlite> .mode line
sqlite> select 60 | 13;
60 | 13 = 61

sqlite> select 60 & 13;
60 & 13 = 12

sqlite> select  60 ^ 13;
10 * 20 = 200

sqlite>  select  (~60);
(~60) = -61

sqlite>  select  (60 << 2);
(60 << 2) = 240

sqlite>  select  (60 >> 2);
(60 >> 2) = 15
```

SQLite 表达式

表达式是一个或多个值、运算符和计算值的SQL函数的组合。

SQL 表达式与公式类似，都写在查询语言中。您还可以使用特定的数据集来查询数据库。

语法

假设 SELECT 语句的基本语法如下：

```
SELECT column1, column2, columnN
FROM table_name
WHERE [CONTION | EXPRESSION];
```

有不同类型的 SQLite 表达式，具体讲解如下：

SQLite - 布尔表达式

SQLite 的布尔表达式在匹配单个值的基础上获取数据。语法如下：

```
SELECT column1, column2, columnN
FROM table_name
WHERE SINGLE VALUE MATCHTING EXPRESSION;
```

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面的实例演示了 SQLite 布尔表达式的用法：

```
sqlite> SELECT * FROM COMPANY WHERE SALARY = 10000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
4           James        24          Houston   10000.0
```

SQLite - 数值表达式

这些表达式用来执行查询中的任何数学运算。语法如下：

```
SELECT numerical_expression as  OPERATION_NAME
[FROM table_name WHERE CONDITION] ;
```

在这里，numerical_expression 用于数学表达式或任何公式。下面的实例演示了 SQLite 数值表达式的用法：

```
sqlite> SELECT (15 + 6) AS ADDITION
ADDITION = 21
```

有几个内置的函数，比如 avg()、sum()、count()，等等，执行被称为对一个表或一个特定的表列的汇总数据计算。

```
sqlite> SELECT COUNT(*) AS "RECORDS" FROM COMPANY;
RECORDS = 7
```

SQLite - 日期表达式

日期表达式返回当前系统日期和时间值，这些表达式将被用于各种数据操作。

```
sqlite>  SELECT CURRENT_TIMESTAMP;
CURRENT_TIMESTAMP = 2013-03-17 10:43:35
```

SQLite Where 子句

SQLite的 **WHERE** 子句用于指定从一个表或多个表中获取数据的条件。

如果满足给定的条件，即为真（true）时，则从表中返回特定的值。您可以使用 WHERE 子句来过滤记录，只获取需要的记录。

WHERE 子句不仅可用在 SELECT 语句中，它也可用在 UPDATE、DELETE 语句中，等等，这些我们将在随后的章节中学习到。

语法

SQLite 的带有 WHERE 子句的 SELECT 语句的基本语法如下：

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition]
```

实例

您还可以使用比较或逻辑运算符指定条件，比如 >、<、=、LIKE、NOT，等等。假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面的实例演示了 SQLite 逻辑运算符的用法。下面的 SELECT 语句列出了 AGE 大于等于 25 且工资大于等于 65000.00 的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE AGE >= 25 AND SALARY >= 65000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

下面的 SELECT 语句列出了 AGE 大于等于 25 或工资大于等于 65000.00 的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE AGE >= 25 OR SALARY >= 65000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

下面的 SELECT 语句列出了 AGE 不为 NULL 的所有记录，结果显示所有的记录，意味着没有一个记录的 AGE 等于 NULL：

```
sqlite>  SELECT * FROM COMPANY WHERE AGE IS NOT NULL;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面的 SELECT 语句列出了 NAME 以 'Ki' 开始的所有记录，'Ki' 之后的字符不做限制：

```
sqlite> SELECT * FROM COMPANY WHERE NAME LIKE 'Ki%';
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
6           Kim         22          South-Hall  45000.0
```

下面的 SELECT 语句列出了 NAME 以 'Ki' 开始的所有记录，'Ki' 之后的字符不做限制：

```
sqlite> SELECT * FROM COMPANY WHERE NAME GLOB 'Ki*';
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
6           Kim         22          South-Hall  45000.0
```

下面的 SELECT 语句列出了 AGE 的值为 25 或 27 的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE AGE IN ( 25, 27 );
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

下面的 SELECT 语句列出了 AGE 的值既不是 25 也不是 27 的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE AGE NOT IN ( 25, 27 );
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
3           Teddy       23          Norway      20000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面的 SELECT 语句列出了 AGE 的值在 25 与 27 之间的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE AGE BETWEEN 25 AND 27;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

下面的 SELECT 语句使用 SQL 子查询，子查询查找 SALARY > 65000 的带有 AGE 字段的所有记录，后边的 WHERE 子句与 EXISTS 运算符一起使用，列出了外查询中的 AGE 存在于子查询返回的结果中的所有记录：

```
sqlite> SELECT AGE FROM COMPANY
        WHERE EXISTS (SELECT AGE FROM COMPANY WHERE SALARY > 65000)
AGE
----------
32
25
23
25
27
22
24
```

下面的 SELECT 语句使用 SQL 子查询，子查询查找 SALARY > 65000 的带有 AGE 字段的所有记录，后边的 WHERE 子句与 > 运算符一起使用，列出了外查询中的 AGE 大于子查询返回的结果中的年龄的所有记录：

```
sqlite> SELECT * FROM COMPANY
        WHERE AGE > (SELECT AGE FROM COMPANY WHERE SALARY > 65000);
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
```

SQLite AND/OR 运算符

SQLite 的 **AND** 和 **OR** 运算符用于编译多个条件来缩小在 SQLite 语句中所选的数据。这两个运算符被称为连接运算符。

这些运算符为同一个 SQLite 语句中不同的运算符之间的多个比较提供了可能。

AND 运算符

AND 运算符允许在一个 SQL 语句的 WHERE 子句中的多个条件的存在。使用 AND 运算符时，只有当所有条件都为真（true）时，整个条件为真（true）。例如，只有当 condition1 和 condition2 都为真（true）时，[condition1] AND [condition2] 为真（true）。

语法

带有 WHERE 子句的 AND 运算符的基本语法如下：

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition1] AND [condition2]...AND [conditionN];
```

您可以使用 AND 运算符来结合 N 个数量的条件。SQLite 语句需要执行的动作是，无论是事务或查询，所有由 AND 分隔的条件都必须为真（TRUE）。

实例

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面的 SELECT 语句列出了 AGE 大于等于 25 且工资大于等于 65000.00 的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE AGE >= 25 AND SALARY >= 65000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

OR 运算符

OR 运算符也用于结合一个 SQL 语句的 WHERE 子句中的多个条件。使用 OR 运算符时，只要当条件中任何一个为真（true）时，整个条件为真（true）。例如，只要当 condition1 或 condition2 有一个为真（true）时，[condition1] OR [condition2] 为真（true）。

语法

带有 WHERE 子句的 OR 运算符的基本语法如下：

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition1] OR [condition2]...OR [conditionN]
```

您可以使用 OR 运算符来结合 N 个数量的条件。SQLite 语句需要执行的动作是，无论是事务或查询，只要任何一个由 OR 分隔的条件为真（TRUE）即可。

实例

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面的 SELECT 语句列出了 AGE 大于等于 25 或工资大于等于 65000.00 的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE AGE >= 25 OR SALARY >= 65000;
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

SQLite Update 语句

SQLite 的 **UPDATE** 查询用于修改表中已有的记录。可以使用带有 WHERE 子句的 UPDATE 查询来更新选定行，否则所有的行都会被更新。

语法

带有 WHERE 子句的 UPDATE 查询的基本语法如下：

```
UPDATE table_name
SET column1 = value1, column2 = value2...., columnN = valueN
WHERE [condition];
```

您可以使用 AND 或 OR 运算符来结合 N 个数量的条件。

实例

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面是一个实例，它会更新 ID 为 6 的客户地址：

```
sqlite> UPDATE COMPANY SET ADDRESS = 'Texas' WHERE ID = 6;
```

现在，COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          Texas       45000.0
7           James       24          Houston     10000.0
```

如果您想修改 COMPANY 表中 ADDRESS 和 SALARY 列的所有值，则不需要使用 WHERE 子句，UPDATE 查询如下：

```
sqlite> UPDATE COMPANY SET ADDRESS = 'Texas', SALARY = 20000.00;
```

现在，COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          Texas       20000.0
2           Allen       25          Texas       20000.0
3           Teddy       23          Texas       20000.0
4           Mark        25          Texas       20000.0
5           David       27          Texas       20000.0
6           Kim         22          Texas       20000.0
7           James       24          Texas       20000.0
```

SQLite Delete 语句

SQLite 的 **DELETE** 查询用于删除表中已有的记录。可以使用带有 WHERE 子句的 DELETE 查询来删除选定行，否则所有的记录都会被删除。

语法

带有 WHERE 子句的 DELETE 查询的基本语法如下：

```
DELETE FROM table_name
WHERE [condition];
```

您可以使用 AND 或 OR 运算符来结合 N 个数量的条件。

实例

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面是一个实例，它会删除 ID 为 7 的客户：

```
sqlite> DELETE FROM COMPANY WHERE ID = 7;
```

现在，COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
```

如果您想要从 COMPANY 表中删除所有记录，则不需要使用 WHERE 子句，DELETE 查询如下：

```
sqlite> DELETE FROM COMPANY;
```

现在，COMPANY 表中没有任何的记录，因为所有的记录已经通过 DELETE 语句删除。

SQLite Like 子句

SQLite 的 **LIKE** 运算符是用来匹配通配符指定模式的文本值。如果搜索表达式与模式表达式匹配，LIKE 运算符将返回真（true），也就是 1。这里有两个通配符与 LIKE 运算符一起使用：

- 百分号 （%）
- 下划线 （_）

百分号（%）代表零个、一个或多个数字或字符。下划线（_）代表一个单一的数字或字符。这些符号可以被组合使用。

语法

% 和 _ 的基本语法如下：

```
SELECT FROM table_name
WHERE column LIKE 'XXXX%'

or

SELECT FROM table_name
WHERE column LIKE '%XXXX%'

or

SELECT FROM table_name
WHERE column LIKE 'XXXX_'

or

SELECT FROM table_name
WHERE column LIKE '_XXXX'

or

SELECT FROM table_name
WHERE column LIKE '_XXXX_'
```

您可以使用 AND 或 OR 运算符来结合 N 个数量的条件。在这里，XXXX 可以是任何数字或字符串值。

实例

下面一些实例演示了 带有 '%' 和 '_' 运算符的 LIKE 子句不同的地方：

语句	描述
WHERE SALARY LIKE '200%'	查找以 200 开头的任意值
WHERE SALARY LIKE '%200%'	查找任意位置包含 200 的任意值
WHERE SALARY LIKE '_00%'	查找第二位和第三位为 00 的任意值
WHERE SALARY LIKE '2_%_%'	查找以 2 开头，且长度至少为 3 个字符的任意值
WHERE SALARY LIKE '%2'	查找以 2 结尾的任意值
WHERE SALARY LIKE '_2%3'	查找第二位为 2，且以 3 结尾的任意值
WHERE SALARY LIKE '2___3'	查找长度为 5 位数，且以 2 开头以 3 结尾的任意值

让我们举一个实际的例子，假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面是一个实例，它显示 COMPANY 表中 AGE 以 2 开头的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE AGE  LIKE '2%';
```

这将产生以下结果：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面是一个实例，它显示 COMPANY 表中 ADDRESS 文本里包含一个连字符（-）的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE ADDRESS  LIKE '%-%';
```

这将产生以下结果：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
4           Mark        25          Rich-Mond   65000.0
6           Kim         22          South-Hall  45000.0
```

SQLite Glob 子句

SQLite 的 **GLOB** 运算符是用来匹配通配符指定模式的文本值。如果搜索表达式与模式表达式匹配，GLOB 运算符将返回真（true），也就是 1。与 LIKE 运算符不同的是，GLOB 是大小写敏感的，对于下面的通配符，它遵循 UNIX 的语法。

- 星号 （*）
- 问号 （?）

星号（*）代表零个、一个或多个数字或字符。问号（?）代表一个单一的数字或字符。这些符号可以被组合使用。

语法

- 和 ? 的基本语法如下：

```
SELECT FROM table_name
WHERE column GLOB 'XXXX*'

or

SELECT FROM table_name
WHERE column GLOB '*XXXX*'

or

SELECT FROM table_name
WHERE column GLOB 'XXXX?'

or

SELECT FROM table_name
WHERE column GLOB '?XXXX'

or

SELECT FROM table_name
WHERE column GLOB '?XXXX?'

or

SELECT FROM table_name
WHERE column GLOB '????'
```

您可以使用 AND 或 OR 运算符来结合 N 个数量的条件。在这里，XXXX 可以是任何数字或字符串值。

实例

下面一些实例演示了 带有 '*' 和 '?' 运算符的 GLOB 子句不同的地方：

语句	描述
WHERE SALARY GLOB '200*'	查找以 200 开头的任意值
WHERE SALARY GLOB '*200*'	查找任意位置包含 200 的任意值
WHERE SALARY GLOB '?00*'	查找第二位和第三位为 00 的任意值
WHERE SALARY GLOB '2??'	查找以 2 开头，且长度至少为 3 个字符的任意值
WHERE SALARY GLOB '*2'	查找以 2 结尾的任意值
WHERE SALARY GLOB '?2*3'	查找第二位为 2，且以 3 结尾的任意值
WHERE SALARY GLOB '2???3'	查找长度为 5 位数，且以 2 开头以 3 结尾的任意值

让我们举一个实际的例子，假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面是一个实例，它显示 COMPANY 表中 AGE 以 2 开头的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE AGE  GLOB '2*';
```

这将产生以下结果：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面是一个实例，它显示 COMPANY 表中 ADDRESS 文本里包含一个连字符（-）的所有记录：

```
sqlite> SELECT * FROM COMPANY WHERE ADDRESS  GLOB '*-*';
```

这将产生以下结果：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
4           Mark        25          Rich-Mond   65000.0
6           Kim         22          South-Hall  45000.0
```

SQLite Limit 子句

SQLite 的 **LIMIT** 子句用于限制由 SELECT 语句返回的数据数量。

语法

带有 LIMIT 子句的 SELECT 语句的基本语法如下：

```
SELECT column1, column2, columnN
FROM table_name
LIMIT [no of rows]
```

下面是 LIMIT 子句与 OFFSET 子句一起使用时的语法：

```
SELECT column1, column2, columnN
FROM table_name
LIMIT [no of rows] OFFSET [row num]
```

SQLite 引擎将返回从下一行开始直到给定的 OFFSET 为止的所有行，如下面的最后一个实例所示。

实例

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面是一个实例，它限制了您想要从表中提取的行数：

```
sqlite> SELECT * FROM COMPANY LIMIT 6;
```

这将产生以下结果：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
```

但是，在某些情况下，可能需要从一个特定的偏移开始提取记录。下面是一个实例，从第三位开始提取 3 个记录：

```
sqlite> SELECT * FROM COMPANY LIMIT 3 OFFSET 2;
```

这将产生以下结果：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

SQLite Order By

SQLite 的 **ORDER BY** 子句是用来基于一个或多个列按升序或降序顺序排列数据。

语法

ORDER BY 子句的基本语法如下：

```
SELECT column-list
FROM table_name
[WHERE condition]
[ORDER BY column1, column2, .. columnN] [ASC | DESC];
```

您可以在 ORDER BY 子句中使用多个列。确保您使用的排序列在列清单中。

实例

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面是一个实例，它会将结果按 SALARY 降序排序：

```
sqlite> SELECT * FROM COMPANY ORDER BY SALARY ASC;
```

这将产生以下结果：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
7           James       24          Houston     10000.0
2           Allen       25          Texas       15000.0
1           Paul        32          California  20000.0
3           Teddy       23          Norway      20000.0
6           Kim         22          South-Hall  45000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

下面是一个实例，它会将结果按 NAME 和 SALARY 降序排序：

```
sqlite> SELECT * FROM COMPANY ORDER BY NAME, SALARY ASC;
```

这将产生以下结果：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
5           David       27          Texas       85000.0
7           James       24          Houston     10000.0
6           Kim         22          South-Hall  45000.0
4           Mark        25          Rich-Mond   65000.0
1           Paul        32          California  20000.0
3           Teddy       23          Norway      20000.0
```

下面是一个实例，它会将结果按 NAME 降序排序：

```
sqlite> SELECT * FROM COMPANY ORDER BY NAME DESC;
```

这将产生以下结果：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
3           Teddy       23          Norway      20000.0
1           Paul        32          California  20000.0
4           Mark        25          Rich-Mond   65000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
5           David       27          Texas       85000.0
2           Allen       25          Texas       15000.0
```

SQLite Group By

SQLite 的 **GROUP BY** 子句用于与 SELECT 语句一起使用，来对相同的数据进行分组。

在 SELECT 语句中，GROUP BY 子句放在 WHERE 子句之后，放在 ORDER BY 子句之前。

语法

下面给出了 GROUP BY 子句的基本语法。GROUP BY 子句必须放在 WHERE 子句中的条件之后，必须放在 ORDER BY 子句之前。

```
SELECT column-list
FROM table_name
WHERE [ conditions ]
GROUP BY column1, column2....columnN
ORDER BY column1, column2....columnN
```

您可以在 GROUP BY 子句中使用多个列。确保您使用的分组列在列清单中。

实例

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

如果您想了解每个客户的工资总额，则可使用 GROUP BY 查询，如下所示：

```
sqlite> SELECT NAME, SUM(SALARY) FROM COMPANY GROUP BY NAME;
```

这将产生以下结果：

```
NAME        SUM(SALARY)
----------  -----------
Allen       15000.0
David       85000.0
James       10000.0
Kim         45000.0
Mark        65000.0
Paul        20000.0
Teddy       20000.0
```

现在，让我们使用下面的 INSERT 语句在 COMPANY 表中另外创建三个记录：

```
INSERT INTO COMPANY VALUES (8, 'Paul', 24, 'Houston', 20000.00 );
INSERT INTO COMPANY VALUES (9, 'James', 44, 'Norway', 5000.00 );
INSERT INTO COMPANY VALUES (10, 'James', 45, 'Texas', 5000.00 );
```

现在，我们的表具有重复名称的记录，如下所示：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
8           Paul        24          Houston     20000.0
9           James       44          Norway      5000.0
10          James       45          Texas       5000.0
```

让我们用同样的 GROUP BY 语句来对所有记录按 NAME 列进行分组，如下所示：

```
sqlite> SELECT NAME, SUM(SALARY) FROM COMPANY GROUP BY NAME ORDER E
```

这将产生以下结果：

```
NAME        SUM(SALARY)
----------  -----------
Allen       15000
David       85000
James       20000
Kim         45000
Mark        65000
Paul        40000
Teddy       20000
```

让我们把 ORDER BY 子句与 GROUP BY 子句一起使用，如下所示：

```
sqlite>  SELECT NAME, SUM(SALARY)
         FROM COMPANY GROUP BY NAME ORDER BY NAME DESC;
```

这将产生以下结果：

```
NAME        SUM(SALARY)
----------  -----------
Teddy       20000
Paul        40000
Mark        65000
Kim         45000
James       20000
David       85000
Allen       15000
```

SQLite Having 子句

HAVING 子句允许指定条件来过滤将出现在最终结果中的分组结果。

WHERE 子句在所选列上设置条件，而 HAVING 子句则在由 GROUP BY 子句创建的分组上设置条件。

语法

下面是 HAVING 子句在 SELECT 查询中的位置：

```
SELECT
FROM
WHERE
GROUP BY
HAVING
ORDER BY
```

在一个查询中，HAVING 子句必须放在 GROUP BY 子句之后，必须放在 ORDER BY 子句之前。下面是包含 HAVING 子句的 SELECT 语句的语法：

```
SELECT column1, column2
FROM table1, table2
WHERE [ conditions ]
GROUP BY column1, column2
HAVING [ conditions ]
ORDER BY column1, column2
```

实例

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
8           Paul        24          Houston     20000.0
9           James       44          Norway      5000.0
10          James       45          Texas       5000.0
```

下面是一个实例，它将显示名称计数小于 2 的所有记录：

```
sqlite > SELECT * FROM COMPANY GROUP BY name HAVING count(name) < 2
```

这将产生以下结果：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000
5           David       27          Texas       85000
6           Kim         22          South-Hall  45000
4           Mark        25          Rich-Mond   65000
3           Teddy       23          Norway      20000
```

下面是一个实例，它将显示名称计数大于 2 的所有记录：

```
sqlite > SELECT * FROM COMPANY GROUP BY name HAVING count(name) > 2
```

这将产生以下结果：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
10          James       45          Texas       5000
```

SQLite Distinct 关键字

SQLite 的 **DISTINCT** 关键字与 SELECT 语句一起使用，来消除所有重复的记录，并只获取唯一一次记录。

有可能出现一种情况，在一个表中有多个重复的记录。当提取这样的记录时，DISTINCT 关键字就显得特别有意义，它只获取唯一一次记录，而不是获取重复记录。

语法

用于消除重复记录的 DISTINCT 关键字的基本语法如下：

```
SELECT DISTINCT column1, column2,.....columnN
FROM table_name
WHERE [condition]
```

实例

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
8           Paul        24          Houston     20000.0
9           James       44          Norway      5000.0
10          James       45          Texas       5000.0
```

首先，让我们来看看下面的 SELECT 查询，它将返回重复的工资记录：

```
sqlite> SELECT name FROM COMPANY;
```

这将产生以下结果：

```
NAME
----------
Paul
Allen
Teddy
Mark
David
Kim
James
Paul
James
James
```

现在，让我们在上述的 SELECT 查询中使用 **DISTINCT** 关键字：

```
sqlite> SELECT DISTINCT name FROM COMPANY;
```

这将产生以下结果，没有任何重复的条目：

```
NAME
----------
Paul
Allen
Teddy
Mark
David
Kim
James
```

SQLite 基础

SQLite PRAGMA

SQLite 的 **PRAGMA** 命令是一个特殊的命令，可以用在 SQLite 环境内控制各种环境变量和状态标志。一个 PRAGMA 值可以被读取，也可以根据需求进行设置。

语法

要查询当前的 PRAGMA 值，只需要提供该 pragma 的名字：

```
PRAGMA pragma_name;
```

要为 PRAGMA 设置一个新的值，语法如下：

```
PRAGMA pragma_name = value;
```

设置模式，可以是名称或等值的整数，但返回的值将始终是一个整数。

auto_vacuum Pragma

auto_vacuum Pragma 获取或设置 auto-vacuum 模式。语法如下：

```
PRAGMA [database.]auto_vacuum;
PRAGMA [database.]auto_vacuum = mode;
```

其中，**mode** 可以是以下任何一种：

Pragma 值	描述
0 或 NONE	禁用 Auto-vacuum。这是默认模式，意味着数据库文件尺寸大小不会缩小，除非手动使用 VACUUM 命令。
1 或 FULL	启用 Auto-vacuum，是全自动的。在该模式下，允许数据库文件随着数据从数据库移除而缩小。
2 或 INCREMENTAL	启用 Auto-vacuum，但是必须手动激活。在该模式下，引用数据被维持，免费页面只放在免费列表中。这些页面可在任何时候使用 **incremental_vacuum pragma** 进行覆盖。

cache_size Pragma

cache_size Pragma 可获取或暂时设置在内存中页面缓存的最大尺寸。语法如下：

```
PRAGMA [database.]cache_size;
PRAGMA [database.]cache_size = pages;
```

pages 值表示在缓存中的页面数。内置页面缓存的默认大小为 2,000 页，最小尺寸为 10 页。

case_sensitive_like Pragma

case_sensitive_like Pragma 控制内置的 LIKE 表达式的大小写敏感度。默认情况下，该 Pragma 为 false，这意味着，内置的 LIKE 操作符忽略字母的大小写。语法如下：

```
PRAGMA case_sensitive_like = [true|false];
```

目前没有办法查询该 Pragma 的当前状态。

count_changes Pragma

count_changes Pragma 获取或设置数据操作语句的返回值，如 INSERT、UPDATE 和 DELETE。语法如下：

```
PRAGMA count_changes;
PRAGMA count_changes = [true|false];
```

默认情况下，该 Pragma 为 false，这些语句不返回任何东西。如果设置为 true，每个所提到的语句将返回一个单行单列的表，由一个单一的整数值组成，该整数表示操作影响的行。

database_list Pragma

database_list Pragma 将用于列出了所有的数据库连接。语法如下：

```
PRAGMA database_list;
```

该 Pragma 将返回一个单行三列的表格，每当打开或附加数据库时，会给出数据库中的序列号，它的名称和相关的文件。

encoding Pragma

encoding Pragma 控制字符串如何编码及存储在数据库文件中。语法如下：

```
PRAGMA encoding;
PRAGMA encoding = format;
```

格式值可以是 UTF-8、UTF-16le 或 UTF-16be 之一。

freelist_count Pragma

freelist_count Pragma 返回一个整数，表示当前被标记为免费和可用的数据库页数。语法如下：

```
PRAGMA [database.]freelist_count;
```

格式值可以是 UTF-8、UTF-16le 或 UTF-16be 之一。

index_info Pragma

index_info Pragma 返回关于数据库索引的信息。语法如下：

```
PRAGMA [database.]index_info( index_name );
```

结果集将为每个包含在给出列序列的索引、表格内的列索引、列名称的列显示一行。

index_list Pragma

index_list Pragma 列出所有与表相关联的索引。语法如下：

```
PRAGMA [database.]index_list( table_name );
```

结果集将为每个给出列序列的索引、索引名称、表示索引是否唯一的标识显示一行。

journal_mode Pragma

journal_mode Pragma 获取或设置控制日志文件如何存储和处理的日志模式。语法如下：:

```
PRAGMA journal_mode;
PRAGMA journal_mode = mode;
PRAGMA database.journal_mode;
PRAGMA database.journal_mode = mode;
```

这里支持五种日志模式：

Pragma 值	描述
DELETE	默认模式。在该模式下，在事务结束时，日志文件将被删除。
TRUNCATE	日志文件被阶段为零字节长度。
PERSIST	日志文件被留在原地，但头部被重写，表明日志不再有效。
MEMORY	日志记录保留在内存中，而不是磁盘上。
OFF	不保留任何日志记录。

max_page_count Pragma

max_page_count Pragma 为数据库获取或设置允许的最大页数。语法如下：

```
PRAGMA [database.]max_page_count;
PRAGMA [database.]max_page_count = max_page;
```

默认值是 1,073,741,823，这是一个千兆的页面，即如果默认 1 KB 的页面大小，那么数据库中增长起来的一个兆字节。

page_count Pragma

page_count Pragma 返回当前数据库中的网页数量。语法如下：

```
PRAGMA [database.]page_count;
```

数据库文件的大小应该是 page_count * page_size。

page_size Pragma

page_size Pragma 获取或设置数据库页面的大小。语法如下：

```
PRAGMA [database.]page_size;
PRAGMA [database.]page_size = bytes;
```

默认情况下，允许的尺寸是 512、1024、2048、4096、8192、16384、32768 字节。改变现有数据库页面大小的唯一方法就是设置页面大小，然后立即 VACUUM 该数据库。

parser_trace Pragma

parser_trace Pragma 随着它解析 SQL 命令来控制打印的调试状态，语法如下：

```
PRAGMA parser_trace = [true|false];
```

默认情况下，它被设置为 false，但设置为 true 时则启用，此时 SQL 解析器会随着它解析 SQL 命令来打印出它的状态。

recursive_triggers Pragma

recursive_triggers Pragma 获取或设置递归触发器功能。如果未启用递归触发器，一个触发动作将不会触发另一个触发。语法如下：

```
PRAGMA recursive_triggers;
PRAGMA recursive_triggers = [true|false];
```

schema_version Pragma

schema_version Pragma 获取或设置存储在数据库头中的的架构版本值。语法如下：

```
PRAGMA [database.]schema_version;
PRAGMA [database.]schema_version = number;
```

这是一个 32 位有符号整数值，用来跟踪架构的变化。每当一个架构改变命令执行（比如 CREATE... 或 DROP...）时，这个值会递增。

secure_delete Pragma

secure_delete Pragma 用来控制内容是如何从数据库中删除。语法如下：

```
PRAGMA secure_delete;
PRAGMA secure_delete = [true|false];
PRAGMA database.secure_delete;
PRAGMA database.secure_delete = [true|false];
```

安全删除标志的默认值通常是关闭的，但是这是可以通过 SQLITE_SECURE_DELETE 构建选项来改变的。

sql_trace Pragma

sql_trace Pragma 用于把 SQL 跟踪结果转储到屏幕上。语法如下：

```
PRAGMA sql_trace;
PRAGMA sql_trace = [true|false];
```

SQLite 必须通过 SQLITE_DEBUG 指令来编译要引用的该 Pragma。

synchronous Pragma

synchronous Pragma 获取或设置当前磁盘的同步模式，该模式控制积极的 SQLite 如何将数据写入物理存储。语法如下：

```
PRAGMA [database.]synchronous;
PRAGMA [database.]synchronous = mode;
```

SQLite 支持下列同步模式：

Pragma 值	描述
0 或 OFF	不进行同步。
1 或 NORMAL	在关键的磁盘操作的每个序列后同步。
2 或 FULL	在每个关键的磁盘操作后同步。

temp_store Pragma

temp_store Pragma 获取或设置临时数据库文件所使用的存储模式。语法如下：

```
PRAGMA temp_store;
PRAGMA temp_store = mode;
```

SQLite 支持下列存储模式：

Pragma 值	描述
0 或 DEFAULT	默认使用编译时的模式。通常是 FILE。
1 或 FILE	使用基于文件的存储。
2 或 MEMORY	使用基于内存的存储。

temp_store_directory Pragma

temp_store_directory Pragma 获取或设置用于临时数据库文件的位置。语法如下：

```
PRAGMA temp_store_directory;
PRAGMA temp_store_directory = 'directory_path';
```

user_version Pragma

user_version Pragma 获取或设置存储在数据库头的用户自定义的版本值。语法如下：

```
PRAGMA [database.]user_version;
PRAGMA [database.]user_version = number;
```

这是一个 32 位的有符号整数值，可以由开发人员设置，用于版本跟踪的目的。

writable_schema Pragma

writable_schema Pragma 获取或设置是否能够修改系统表。语法如下：

```
PRAGMA writable_schema;
PRAGMA writable_schema = [true|false];
```

如果设置了该 Pragma，则表以 sqlite_ 开始，可以创建和修改，包括 sqlite_master 表。使用该 Pragma 时要注意，因为它可能导致整个数据库损坏。

SQLite 约束

约束是在表的数据列上强制执行的规则。这些是用来限制可以插入到表中的数据类型。这确保了数据库中数据的准确性和可靠性。

约束可以是列级或表级。列级约束仅适用于列，表级约束被应用到整个表。

以下是在 SQLite 中常用的约束。

- **NOT NULL** 约束：确保某列不能有 NULL 值。
- **DEFAULT** 约束：当某列没有指定值时，为该列提供默认值。
- **UNIQUE** 约束：确保某列中的所有值是不同的。
- **PRIMARY Key** 约束：唯一标识数据库表中的各行/记录。
- **CHECK** 约束：CHECK 约束确保某列中的所有值满足一定条件。

NOT NULL 约束

默认情况下，列可以保存 NULL 值。如果您不想某列有 NULL 值，那么需要在该列上定义此约束，指定在该列上不允许 NULL 值。

NULL 与没有数据是不一样的，它代表着未知的数据。

实例

例如，下面的 SQLite 语句创建一个新的表 COMPANY，并增加了五列，其中 ID、NAME 和 AGE 三列指定不接受 NULL 值：

```
CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

DEFAULT 约束

DEFAULT 约束在 INSERT INTO 语句没有提供一个特定的值时，为列提供一个默认值。

实例

例如，下面的 SQLite 语句创建一个新的表 COMPANY，并增加了五列。在这里，SALARY 列默认设置为 5000.00。所以当 INSERT INTO 语句没有为该列提供值时，该列将被设置为 5000.00。

```
CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL    DEFAULT 50000.00
);
```

UNIQUE 约束

UNIQUE 约束防止在一个特定的列存在两个记录具有相同的值。在 COMPANY 表中，例如，您可能要防止两个或两个以上的人具有相同的年龄。

实例

例如，下面的 SQLite 语句创建一个新的表 COMPANY，并增加了五列。在这里，AGE 列设置为 UNIQUE，所以不能有两个相同年龄的记录：

```
CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL UNIQUE,
   ADDRESS        CHAR(50),
   SALARY         REAL    DEFAULT 50000.00
);
```

PRIMARY KEY 约束

PRIMARY KEY 约束唯一标识数据库表中的每个记录。在一个表中可以有多个 UNIQUE 列，但只能有一个主键。在设计数据库表时，主键是很重要的。主键是唯一的 ID。

我们使用主键来引用表中的行。可通过把主键设置为其他表的外键，来创建表之间的关系。由于"长期存在编码监督"，在 SQLite 中，主键可以是 NULL，这是与其他数据库不同的地方。

主键是表中的一个字段，唯一标识数据库表中的各行/记录。主键必须包含唯一值。主键列不能有 NULL 值。

一个表只能有一个主键，它可以由一个或多个字段组成。当多个字段作为主键，它们被称为复合键。

如果一个表在任何字段上定义了一个主键，那么在这些字段上不能有两个记录具有相同的值。

实例

已经看到了我们创建以 ID 作为主键的 COMAPNY 表的各种实例：

```
CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

CHECK 约束

CHECK 约束启用输入一条记录要检查值的条件。如果条件值为 false，则记录违反了约束，且不能输入到表。

实例

例如，下面的 SQLite 创建一个新的表 COMPANY，并增加了五列。在这里，我们为 SALARY 列添加 CHECK，所以工资不能为零：

```
CREATE TABLE COMPANY3(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL    CHECK(SALARY > 0)
);
```

删除约束

SQLite 支持 ALTER TABLE 的有限子集。在 SQLite 中，ALTER TABLE 命令允许用户重命名表，或向现有表添加一个新的列。重命名列，删除一列，或从一个表中添加或删除约束都是不可能的。

SQLite Joins

SQLite 的 **Joins** 子句用于结合两个或多个数据库中表的记录。JOIN 是一种通过共同值来结合两个表中字段的手段。

SQL 定义了三种主要类型的连接：

- 交叉连接 - CROSS JOIN
- 内连接 - INNER JOIN
- 外连接 - OUTER JOIN

在我们继续之前，让我们假设有两个表 COMPANY 和 DEPARTMENT。我们已经看到了用来填充 COMPANY 表的 INSERT 语句。现在让我们假设 COMPANY 表的记录列表如下：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

另一个表是 DEPARTMENT，定义如下：

```
CREATE TABLE DEPARTMENT(
   ID INT PRIMARY KEY      NOT NULL,
   DEPT           CHAR(50) NOT NULL,
   EMP_ID         INT      NOT NULL
);
```

下面是填充 DEPARTMENT 表的 INSERT 语句：

```
INSERT INTO DEPARTMENT (ID, DEPT, EMP_ID)
VALUES (1, 'IT Billing', 1 );

INSERT INTO DEPARTMENT (ID, DEPT, EMP_ID)
VALUES (2, 'Engineering', 2 );

INSERT INTO DEPARTMENT (ID, DEPT, EMP_ID)
VALUES (3, 'Finance', 7 );
```

最后，我们在 DEPARTMENT 表中有下列的记录列表：

```
ID          DEPT        EMP_ID
----------  ----------  ----------
1           IT Billing  1
2           Engineerin  2
3           Finance     7
```

交叉连接 - CROSS JOIN

交叉连接（CROSS JOIN）把第一个表的每一行与第二个表的每一行进行匹配。如果两个输入表分别有 x 和 y 列，则结果表有 x+y 列。由于交叉连接（CROSS JOIN）有可能产生非常大的表，使用时必须谨慎，只在适当的时候使用它们。

下面是交叉连接（CROSS JOIN）的语法：

```
SELECT ... FROM table1 CROSS JOIN table2 ...
```

基于上面的表，我们可以写一个交叉连接（CROSS JOIN），如下所示：

```
sqlite> SELECT EMP_ID, NAME, DEPT FROM COMPANY CROSS JOIN DEPARTMEN
```

上面的查询会产生以下结果：

```
EMP_ID      NAME        DEPT
----------  ----------  ----------
1           Paul        IT Billing
2           Paul        Engineerin
7           Paul        Finance
1           Allen       IT Billing
2           Allen       Engineerin
7           Allen       Finance
1           Teddy       IT Billing
2           Teddy       Engineerin
7           Teddy       Finance
1           Mark        IT Billing
2           Mark        Engineerin
7           Mark        Finance
1           David       IT Billing
2           David       Engineerin
7           David       Finance
1           Kim         IT Billing
2           Kim         Engineerin
7           Kim         Finance
1           James       IT Billing
2           James       Engineerin
7           James       Finance
```

内连接 - INNER JOIN

内连接（INNER JOIN）根据连接谓词结合两个表（table1 和 table2）的列值来创建一个新的结果表。查询会把 table1 中的每一行与 table2 中的每一行进行比较，找到所有满足连接谓词的行的匹配对。当满足连接谓词时，A 和 B 行的每个匹配对的列值会合并成一个结果行。

内连接（INNER JOIN）是最常见的连接类型，是默认的连接类型。INNER 关键字是可选的。

下面是内连接（INNER JOIN）的语法：

```
SELECT ... FROM table1 [INNER] JOIN table2 ON conditional_expressio
```

为了避免冗余，并保持较短的措辞，可以使用 **USING** 表达式声明内连接（INNER JOIN）条件。这个表达式指定一个或多个列的列表：

```
SELECT ... FROM table1 JOIN table2 USING ( column1 ,... ) ...
```

自然连接（NATURAL JOIN）类似于 **JOIN...USING**，只是它会自动测试存在两个表中的每一列的值之间相等值：

```
SELECT ... FROM table1 NATURAL JOIN table2...
```

基于上面的表，我们可以写一个内连接（INNER JOIN），如下所示：

```
sqlite> SELECT EMP_ID, NAME, DEPT FROM COMPANY INNER JOIN DEPARTMEN
        ON COMPANY.ID = DEPARTMENT.EMP_ID;
```

上面的查询会产生以下结果：

```
EMP_ID      NAME        DEPT
----------  ----------  ----------
1           Paul        IT Billing
2           Allen       Engineerin
7           James       Finance
```

外连接 - OUTER JOIN

外连接（OUTER JOIN）是内连接（INNER JOIN）的扩展。虽然 SQL 标准定义了三种类型的外连接：LEFT、RIGHT、FULL，但 SQLite 只支持 左外连接（**LEFT OUTER JOIN**）。

外连接（OUTER JOIN）声明条件的方法与内连接（INNER JOIN）是相同的，使用 ON、USING 或 NATURAL 关键字来表达。最初的结果表以相同的方式进行计算。一旦主连接计算完成，外连接（OUTER JOIN）将从一个或两个表中任何未连接的行合并进来，外连接的列使用 NULL 值，将它们附加到结果表中。

下面是左外连接（LEFT OUTER JOIN）的语法：

```
SELECT ... FROM table1 LEFT OUTER JOIN table2 ON conditional_expres
```

为了避免冗余，并保持较短的措辞，可以使用 **USING** 表达式声明外连接（OUTER JOIN）条件。这个表达式指定一个或多个列的列表：

```
SELECT ... FROM table1 LEFT OUTER JOIN table2 USING ( column1 ,...
```

基于上面的表，我们可以写一个外连接（OUTER JOIN），如下所示：

```
sqlite> SELECT EMP_ID, NAME, DEPT FROM COMPANY LEFT OUTER JOIN DEPA
        ON COMPANY.ID = DEPARTMENT.EMP_ID;
```

上面的查询会产生以下结果：

```
EMP_ID      NAME        DEPT
----------  ----------  ----------
1           Paul        IT Billing
2           Allen       Engineerin
            Teddy
            Mark
            David
            Kim
7           James       Finance
```

SQLite Unions 子句

SQLite的 **UNION** 子句/运算符用于合并两个或多个 SELECT 语句的结果，不返回任何重复的行。

为了使用 UNION，每个 SELECT 被选择的列数必须是相同的，相同数目的列表达式，相同的数据类型，并确保它们有相同的顺序，但它们不必具有相同的长度。

语法

UNION 的基本语法如下：

```
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]

UNION

SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
```

这里给定的条件根据需要可以是任何表达式。

实例

假设有下面两个表，（1）COMPANY 表如下所示：

```
sqlite> select * from COMPANY;
ID          NAME                  AGE         ADDRESS     SALARY
----------  --------------------  ----------  ----------  ---------
1           Paul                  32          California  20000.0
2           Allen                 25          Texas       15000.0
3           Teddy                 23          Norway      20000.0
4           Mark                  25          Rich-Mond   65000.0
5           David                 27          Texas       85000.0
6           Kim                   22          South-Hall  45000.0
7           James                 24          Houston     10000.0
```

（2）另一个表是 DEPARTMENT，如下所示：

```
ID          DEPT                  EMP_ID
----------  --------------------  ----------
1           IT Billing            1
2           Engineering           2
3           Finance               7
4           Engineering           3
5           Finance               4
6           Engineering           5
7           Finance               6
```

现在，让我们使用 SELECT 语句及 UNION 子句来连接两个表，如下所示：

```
sqlite> SELECT EMP_ID, NAME, DEPT FROM COMPANY INNER JOIN DEPARTMEN
        ON COMPANY.ID = DEPARTMENT.EMP_ID
   UNION
     SELECT EMP_ID, NAME, DEPT FROM COMPANY LEFT OUTER JOIN DEPARTM
        ON COMPANY.ID = DEPARTMENT.EMP_ID;
```

这将产生以下结果：

```
EMP_ID      NAME                  DEPT
----------  --------------------  ----------
1           Paul                  IT Billing
2           Allen                 Engineerin
3           Teddy                 Engineerin
4           Mark                  Finance
5           David                 Engineerin
6           Kim                   Finance
7           James                 Finance
```

UNION ALL 子句

UNION ALL 运算符用于结合两个 SELECT 语句的结果，包括重复行。

适用于 UNION 的规则同样适用于 UNION ALL 运算符。

语法

UNION ALL 的基本语法如下：

```
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]

UNION ALL

SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
```

这里给定的条件根据需要可以是任何表达式。

实例

现在，让我们使用 SELECT 语句及 UNION ALL 子句来连接两个表，如下所示：

```
sqlite> SELECT EMP_ID, NAME, DEPT FROM COMPANY INNER JOIN DEPARTMEN
        ON COMPANY.ID = DEPARTMENT.EMP_ID
   UNION ALL
     SELECT EMP_ID, NAME, DEPT FROM COMPANY LEFT OUTER JOIN DEPARTM
        ON COMPANY.ID = DEPARTMENT.EMP_ID;
```

这将产生以下结果：

```
EMP_ID      NAME                  DEPT
----------  --------------------  ----------
1           Paul                  IT Billing
2           Allen                 Engineerin
3           Teddy                 Engineerin
4           Mark                  Finance
5           David                 Engineerin
6           Kim                   Finance
7           James                 Finance
1           Paul                  IT Billing
2           Allen                 Engineerin
3           Teddy                 Engineerin
4           Mark                  Finance
5           David                 Engineerin
6           Kim                   Finance
7           James                 Finance
```

SQLite NULL 值

SQLite 的 **NULL** 是用来表示一个缺失值的项。表中的一个 NULL 值是在字段中显示为空白的一个值。

带有 NULL 值的字段是一个不带有值的字段。NULL 值与零值或包含空格的字段是不同的，理解这点是非常重要的。

语法

创建表时使用 **NULL** 的基本语法如下：

```
SQLite> CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

在这里，**NOT NULL** 表示列总是接受给定数据类型的显式值。这里有两个列我们没有使用 NOT NULL，这意味着这两个列不能为 NULL。

带有 NULL 值的字段在记录创建的时候可以保留为空。

实例

NULL 值在选择数据时会引起问题，因为当把一个未知的值与另一个值进行比较时，结果总是未知的，且不会包含在最后的结果中。假设有下面的表，COMPANY 的记录如下所示：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

让我们使用 UPDATE 语句来设置一些允许空值的值为 NULL，如下所示：

```
sqlite> UPDATE COMPANY SET ADDRESS = NULL, SALARY = NULL where ID I
```

现在，COMPANY 表的记录如下所示：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22
7           James       24
```

接下来，让我们看看 **IS NOT NULL** 运算符的用法，它用来列出所有 SALARY 不为 NULL 的记录：

```
sqlite> SELECT  ID, NAME, AGE, ADDRESS, SALARY
        FROM COMPANY
        WHERE SALARY IS NOT NULL;
```

上面的 SQLite 语句将产生下面的结果：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

下面是 **IS NULL** 运算符的用法，将列出所有 SALARY 为 NULL 的记录：

```
sqlite> SELECT  ID, NAME, AGE, ADDRESS, SALARY
        FROM COMPANY
        WHERE SALARY IS NULL;
```

上面的 SQLite 语句将产生下面的结果：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
6           Kim         22
7           James       24
```

SQLite 别名

您可以暂时把表或列重命名为另一个名字，这被称为别名。使用表别名是指在一个特定的 SQLite 语句中重命名表。重命名是临时的改变，在数据库中实际的表的名称不会改变。

列别名用来为某个特定的 SQLite 语句重命名表中的列。

语法

表 别名的基本语法如下：

```
SELECT column1, column2....
FROM table_name AS alias_name
WHERE [condition];
```

列 别名的基本语法如下：

```
SELECT column_name AS alias_name
FROM table_name
WHERE [condition];
```

实例

假设有下面两个表，（1）COMPANY 表如下所示：

```
sqlite> select * from COMPANY;
ID          NAME                  AGE         ADDRESS     SALARY
----------  --------------------  ----------  ----------  ---------
1           Paul                  32          California  20000.0
2           Allen                 25          Texas       15000.0
3           Teddy                 23          Norway      20000.0
4           Mark                  25          Rich-Mond   65000.0
5           David                 27          Texas       85000.0
6           Kim                   22          South-Hall  45000.0
7           James                 24          Houston     10000.0
```

（2）另一个表是 DEPARTMENT，如下所示：

```
ID          DEPT                  EMP_ID
----------  --------------------  ----------
1           IT Billing            1
2           Engineering           2
3           Finance               7
4           Engineering           3
5           Finance               4
6           Engineering           5
7           Finance               6
```

现在，下面是 表别名 的用法，在这里我们使用 C 和 D 分别作为 COMPANY 和 DEPARTMENT 表的别名：

```
sqlite> SELECT C.ID, C.NAME, C.AGE, D.DEPT
        FROM COMPANY AS C, DEPARTMENT AS D
        WHERE  C.ID = D.EMP_ID;
```

上面的 SQLite 语句将产生下面的结果：

```
ID          NAME        AGE         DEPT
----------  ----------  ----------  ----------
1           Paul        32          IT Billing
2           Allen       25          Engineerin
3           Teddy       23          Engineerin
4           Mark        25          Finance
5           David       27          Engineerin
6           Kim         22          Finance
7           James       24          Finance
```

让我们看一个 列别名 的实例，在这里 COMPANY_ID 是 ID 列的别名，COMPANY_NAME 是 name 列的别名：

```
sqlite> SELECT C.ID AS COMPANY_ID, C.NAME AS COMPANY_NAME, C.AGE, D
        FROM COMPANY AS C, DEPARTMENT AS D
        WHERE  C.ID = D.EMP_ID;
```

上面的 SQLite 语句将产生下面的结果：

```
COMPANY_ID  COMPANY_NAME  AGE         DEPT
----------  ------------  ----------  ----------
1           Paul          32          IT Billing
2           Allen         25          Engineerin
3           Teddy         23          Engineerin
4           Mark          25          Finance
5           David         27          Engineerin
6           Kim           22          Finance
7           James         24          Finance
```

SQLite 触发器（Trigger）

SQLite 的触发器是数据库的回调函数，它会自动执行/指定的数据库事件发生时调用。以下是关于SQLite的触发器的要点： SQLite 触发器（**Trigger**）是数据库的回调函数，它会在指定的数据库事件发生时自动执行/调用。以下是关于 SQLite 的触发器（Trigger）的要点：

- SQLite 的触发器（Trigger）可以指定在特定的数据库表发生 DELETE、INSERT 或 UPDATE 时触发，或在一个或多个指定表的列发生更新时触发。
- SQLite 只支持 FOR EACH ROW 触发器（Trigger），没有 FOR EACH STATEMENT 触发器（Trigger）。因此，明确指定 FOR EACH ROW 是可选的。
- WHEN 子句和触发器（Trigger）动作可能访问使用表单 **NEW.column-name** 和 **OLD.column-name** 的引用插入、删除或更新的行元素，其中 column-name 是从与触发器关联的表的列的名称。
- 如果提供 WHEN 子句，则只针对 WHEN 子句为真的指定行执行 SQL 语句。如果没有提供 WHEN 子句，则针对所有行执行 SQL 语句。
- BEFORE 或 AFTER 关键字决定何时执行触发器动作，决定是在关联行的插入、修改或删除之前或者之后执行触发器动作。
- 当触发器相关联的表删除时，自动删除触发器（Trigger）。
- 要修改的表必须存在于同一数据库中，作为触发器被附加的表或视图，且必须只使用 **tablename**，而不是 **database.tablename**。
- 一个特殊的 SQL 函数 RAISE() 可用于触发器程序内抛出异常。

语法

创建 触发器（**Trigger**） 的基本语法如下：

```
CREATE  TRIGGER trigger_name [BEFORE|AFTER] event_name
ON table_name
BEGIN
 -- Trigger logic goes here....
END;
```

在这里，**event_name** 可以是在所提到的表 **table_name** 上的 *INSERT*、*DELETE* 和 *UPDATE* 数据库操作。您可以在表名后选择指定 FOR EACH ROW。

以下是在 UPDATE 操作上在表的一个或多个指定列上创建触发器（Trigger）的语法：

```
CREATE  TRIGGER trigger_name [BEFORE|AFTER] UPDATE OF column_name
ON table_name
BEGIN
 -- Trigger logic goes here....
END;
```

实例

让我们假设一个情况，我们要为被插入到新创建的 COMPANY 表（如果已经存在，则删除重新创建）中的每一个记录保持审计试验：

```
sqlite> CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

为了保持审计试验，我们将创建一个名为 AUDIT 的新表。每当 COMPANY 表中有一个新的记录项时，日志消息将被插入其中：

```
sqlite> CREATE TABLE AUDIT(
    EMP_ID INT NOT NULL,
    ENTRY_DATE TEXT NOT NULL
);
```

在这里，ID 是 AUDIT 记录的 ID，EMP_ID 是来自 COMPANY 表的 ID，DATE 将保持 COMPANY 中记录被创建时的时间戳。所以，现在让我们在 COMPANY 表上创建一个触发器，如下所示：

```
sqlite> CREATE TRIGGER audit_log AFTER INSERT
ON COMPANY
BEGIN
   INSERT INTO AUDIT(EMP_ID, ENTRY_DATE) VALUES (new.ID, datetime(
END;
```

现在，我们将开始在 COMPANY 表中插入记录，这将导致在 AUDIT 表中创建一个审计日志记录。因此，让我们在 COMPANY 表中创建一个记录，如下所示：

```
sqlite> INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Paul', 32, 'California', 20000.00 );
```

这将在 COMPANY 表中创建如下一个记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
```

同时，将在 AUDIT 表中创建一个记录。这个纪录是触发器的结果，这是我们在 COMPANY 表上的 INSERT 操作上创建的触发器（Trigger）。类似的，可以根据需要在 UPDATE 和 DELETE 操作上创建触发器（Trigger）。

```
EMP_ID      ENTRY_DATE
----------  -------------------
1           2013-04-05 06:26:00
```

列出触发器（TRIGGERS）

您可以从 **sqlite_master** 表中列出所有触发器，如下所示：

```
sqlite> SELECT name FROM sqlite_master
WHERE type = 'trigger';
```

上面的 SQLite 语句只会列出一个条目，如下：

```
name
----------
audit_log
```

如果您想要列出特定表上的触发器，则使用 AND 子句连接表名，如下所示：

```
sqlite> SELECT name FROM sqlite_master
WHERE type = 'trigger' AND tbl_name = 'COMPANY';
```

上面的 SQLite 语句只会列出一个条目，如下：

```
name
----------
audit_log
```

删除触发器（TRIGGERS）

下面是 DROP 命令，可用于删除已有的触发器：

```
sqlite> DROP TRIGGER trigger_name;
```

SQLite 索引（Index）

索引（Index）是一种特殊的查找表，数据库搜索引擎用来加快数据检索。简单地说，索引是一个指向表中数据的指针。一个数据库中的索引与一本书后边的索引是非常相似的。

例如，如果您想在一本讨论某个话题的书中引用所有页面，您首先需要指向索引，索引按字母顺序列出了所有主题，然后指向一个或多个特定的页码。

索引有助于加快 SELECT 查询和 WHERE 子句，但它会减慢使用 UPDATE 和 INSERT 语句时的数据输入。索引可以创建或删除，但不会影响数据。

使用 CREATE INDEX 语句创建索引，它允许命名索引，指定表及要索引的一列或多列，并指示索引是升序排列还是降序排列。

索引也可以是唯一的，与 UNIQUE 约束类似，在列上或列组合上防止重复条目。

CREATE INDEX 命令

CREATE INDEX 的基本语法如下：

```
CREATE INDEX index_name ON table_name;
```

单列索引

单列索引是一个只基于表的一个列上创建的索引。基本语法如下：

```
CREATE INDEX index_name
ON table_name (column_name);
```

唯一索引

使用唯一索引不仅是为了性能，同时也为了数据的完整性。唯一索引不允许任何重复的值插入到表中。基本语法如下：

```
CREATE INDEX index_name
on table_name (column_name);
```

组合索引

组合索引是基于一个表的两个或多个列上创建的索引。基本语法如下：

```
CREATE INDEX index_name
on table_name (column1, column2);
```

是否要创建一个单列索引还是组合索引，要考虑到您在作为查询过滤条件的 WHERE 子句中使用非常频繁的列。

如果值使用到一个列，则选择使用单列索引。如果在作为过滤的 WHERE 子句中有两个或多个列经常使用，则选择使用组合索引。

隐式索引

隐式索引是在创建对象时，由数据库服务器自动创建的索引。索引自动创建为主键约束和唯一约束。

实例

下面是一个例子，我们将在 COMPANY 表的 salary 列上创建一个索引：

```
sqlite> CREATE INDEX salary_index ON COMPANY (salary);
```

现在，让我们使用 **.indices** 命令列出 COMPANY 表上所有可用的索引，如下所示：

```
sqlite> .indices COMPANY
```

这将产生如下结果，其中 *sqlite_autoindex_COMPANY_1* 是创建表时创建的隐式索引。

```
salary_index
sqlite_autoindex_COMPANY_1
```

您可以列出数据库范围的所有索引，如下所示：

```
sqlite> SELECT * FROM sqlite_master WHERE type = 'index';
```

DROP INDEX 命令

一个索引可以使用 SQLite 的 **DROP** 命令删除。当删除索引时应特别注意，因为性能可能会下降或提高。

基本语法如下：

```
DROP INDEX index_name;
```

您可以使用下面的语句来删除之前创建的索引：

```
sqlite> DROP INDEX salary_index;
```

什么情况下要避免使用索引？

虽然索引的目的在于提高数据库的性能，但这里有几个情况需要避免使用索引。使用索引时，应重新考虑下列准则：

- 索引不应该使用在较小的表上。
- 索引不应该使用在有频繁的大批量的更新或插入操作的表上。
- 索引不应该使用在含有大量的 NULL 值的列上。
- 索引不应该使用在频繁操作的列上。

SQLite Indexed By

"INDEXED BY index-name" 子句规定必须需要命名的索引来查找前面表中值。

如果索引名 index-name 不存在或不能用于查询，然后 SQLite 语句的准备失败。

"NOT INDEXED" 子句规定当访问前面的表（包括由 UNIQUE 和 PRIMARY KEY 约束创建的隐式索引）时，没有使用索引。

然而，即使指定了 "NOT INDEXED"，INTEGER PRIMARY KEY 仍然可以被用于查找条目。

语法

下面是 INDEXED BY 子句的语法，它可以与 DELETE、UPDATE 或 SELECT 语句一起使用：

```
SELECT|DELETE|UPDATE column1, column2...
INDEXED BY (index_name)
table_name
WHERE (CONDITION);
```

实例

假设有表 COMPANY，我们将创建一个索引，并用它进行 INDEXED BY 操作。

```
sqlite> CREATE INDEX salary_index ON COMPANY(salary);
sqlite>
```

现在使用 INDEXED BY 子句从表 COMPANY 中选择数据，如下所示：

```
sqlite> SELECT * FROM COMPANY INDEXED BY salary_index WHERE salary
```

SQLite Alter 命令

SQLite 的 **ALTER TABLE** 命令不通过执行一个完整的转储和数据的重载来修改已有的表。您可以使用 ALTER TABLE 语句重命名表，使用 ALTER TABLE 语句还可以在已有的表中添加额外的列。

在 SQLite 中，除了重命名表和在已有的表中添加列，ALTER TABLE 命令不支持其他操作。

语法

用来重命名已有的表的 **ALTER TABLE** 的基本语法如下：

```
ALTER TABLE database_name.table_name RENAME TO new_table_name;
```

用来在已有的表中添加一个新的列的 **ALTER TABLE** 的基本语法如下：

```
ALTER TABLE database_name.table_name ADD COLUMN column_def...;
```

实例

假设我们的 COMPANY 表有如下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

现在，让我们尝试使用 ALTER TABLE 语句重命名该表，如下所示：

```
sqlite> ALTER TABLE COMPANY RENAME TO OLD_COMPANY;
```

上面的 SQLite 语句将重命名 COMPANY 表为 OLD_COMPANY。现在，让我们尝试在 OLD_COMPANY 表中添加一个新的列，如下所示：

```
sqlite> ALTER TABLE OLD_COMPANY ADD COLUMN SEX char(1);
```

现在，COMPANY 表已经改变，使用 SELECT 语句输出如下：

```
ID          NAME        AGE         ADDRESS     SALARY      SEX
----------  ----------  ----------  ----------  ----------  ---
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

请注意，新添加的列是以 NULL 值来填充的。

SQLite Truncate Table

在 SQLite 中，并没有 TRUNCATE TABLE 命令，但可以使用 SQLite 的 **DELETE** 命令从已有的表中删除全部的数据，但建议使用 DROP TABLE 命令删除整个表，然后再重新创建一遍。

语法

DELETE 命令的基本语法如下：

```
sqlite> DELETE FROM table_name;
```

DROP TABLE 的基本语法如下：

```
sqlite> DROP TABLE table_name;
```

如果您使用 DELETE TABLE 命令删除所有记录，建议使用 **VACUUM** 命令清除未使用的空间。

实例

假设 COMPANY 表有如下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

下面为删除上表记录的实例：

```
SQLite> DELETE FROM COMPANY;
SQLite> VACUUM;
```

现在，COMPANY 表中的记录完全被删除，使用 SELECT 语句将没有任何输出。

SQLite 视图（View）

视图（View）只不过是通过相关的名称存储在数据库中的一个 SQLite 语句。视图（View）实际上是一个以预定义的 SQLite 查询形式存在的表的组合。

视图（View）可以包含一个表的所有行或从一个或多个表选定行。视图（View）可以从一个或多个表创建，这取决于要创建视图的 SQLite 查询。、

视图（View）是一种虚表，允许用户实现以下几点：

- 用户或用户组查找结构数据的方式更自然或直观。
- 限制数据访问，用户只能看到有限的数据，而不是完整的表。
- 汇总各种表中的数据，用于生成报告。

SQLite 视图是只读的，因此可能无法在视图上执行 DELETE、INSERT 或 UPDATE 语句。但是可以在视图上创建一个触发器，当尝试 DELETE、INSERT 或 UPDATE 视图时触发，需要做的动作在触发器内容中定义。

创建视图

SQLite 的视图是使用 **CREATE VIEW** 语句创建的。SQLite 视图可以从一个单一的表、多个表或其他视图创建。

CREATE VIEW 的基本语法如下：

```
CREATE [TEMP | TEMPORARY] VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE [condition];
```

您可以在 SELECT 语句中包含多个表，这与在正常的 SQL SELECT 查询中的方式非常相似。如果使用了可选的 TEMP 或 TEMPORARY 关键字，则将在临时数据库中创建视图。

实例

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

现在，下面是一个从 COMPANY 表创建视图的实例。视图只从 COMPANY 表中选取几列：

```
sqlite> CREATE VIEW COMPANY_VIEW AS
SELECT ID, NAME, AGE
FROM  COMPANY;
```

现在，可以查询 COMPANY_VIEW，与查询实际表的方式类似。下面是实例：

```
sqlite> SELECT * FROM COMPANY_VIEW;
```

这将产生以下结果:

```
ID          NAME        AGE
----------  ----------  ----------
1           Paul        32
2           Allen       25
3           Teddy       23
4           Mark        25
5           David       27
6           Kim         22
7           James       24
```

删除视图

要删除视图，只需使用带有 **view_name** 的 DROP VIEW 语句。DROP VIEW 的基本语法如下：

```
sqlite> DROP VIEW view_name;
```

下面的命令将删除我们在前面创建的 COMPANY_VIEW 视图：

```
sqlite> DROP VIEW COMPANY_VIEW;
```

SQLite 事务（Transaction）

事务（Transaction）是一个对数据库执行工作单元。事务（Transaction）是以逻辑顺序完成的工作单位或序列，可以是由用户手动操作完成，也可以是由某种数据库程序自动完成。

事务（Transaction）是指一个或多个更改数据库的扩展。例如，如果您正在创建一个记录或者更新一个记录或者从表中删除一个记录，那么您正在该表上执行事务。重要的是要控制事务以确保数据的完整性和处理数据库错误。

实际上，您可以把许多的 SQLite 查询联合成一组，把所有这些放在一起作为事务的一部分进行执行。

事务的属性

事务（Transaction）具有以下四个标准属性，通常根据首字母缩写为 ACID：

- 原子性（**Atomicity**）：确保工作单位内的所有操作都成功完成，否则，事务会在出现故障时终止，之前的操作也会回滚到以前的状态。
- 一致性（**Consistency)**：确保数据库在成功提交的事务上正确地改变状态。
- 隔离性（**Isolation**）：使事务操作相互独立和透明。
- 持久性（**Durability**）：确保已提交事务的结果或效果在系统发生故障的情况下仍然存在。

事务控制

使用下面的命令来控制事务：

- **BEGIN TRANSACTION**：开始事务处理。
- **COMMIT**：保存更改，或者可以使用 **END TRANSACTION** 命令。
- **ROLLBACK**：回滚所做的更改。

事务控制命令只与 DML 命令 INSERT、UPDATE 和 DELETE 一起使用。他们不能在创建表或删除表时使用，因为这些操作在数据库中是自动提交的。

BEGIN TRANSACTION 命令

事务（Transaction）可以使用 BEGIN TRANSACTION 命令或简单的 BEGIN 命令来启动。此类事务通常会持续执行下去，直到遇到下一个 COMMIT 或 ROLLBACK 命令。不过在数据库关闭或发生错误时，事务处理也会回滚。以下是启动一个事务

的简单语法：

```
BEGIN;

or

BEGIN TRANSACTION;
```

COMMIT 命令

COMMIT 命令是用于把事务调用的更改保存到数据库中的事务命令。

COMMIT 命令把自上次 COMMIT 或 ROLLBACK 命令以来的所有事务保存到数据库。

COMMIT 命令的语法如下：

```
COMMIT;

or

END TRANSACTION;
```

ROLLBACK 命令

ROLLBACK 命令是用于撤消尚未保存到数据库的事务的事务命令。

ROLLBACK 命令只能用于撤销自上次发出 COMMIT 或 ROLLBACK 命令以来的事务。

ROLLBACK 命令的语法如下：

```
ROLLBACK;
```

实例

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

现在，让我们开始一个事务，并从表中删除 age = 25 的记录，最后，我们使用 ROLLBACK 命令撤消所有的更改。

```
sqlite> BEGIN;
sqlite> DELETE FROM COMPANY WHERE AGE = 25;
sqlite> ROLLBACK;
```

检查 COMPANY 表，仍然有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

现在，让我们开始另一个事务，从表中删除 age = 25 的记录，最后我们使用 COMMIT 命令提交所有的更改。

```
sqlite> BEGIN;
sqlite> DELETE FROM COMPANY WHERE AGE = 25;
sqlite> COMMIT;
```

检查 COMPANY 表，有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
3           Teddy       23          Norway      20000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

SQLite 子查询

子查询或内部查询或嵌套查询是在另一个 SQLite 查询内嵌入在 WHERE 子句中的查询。

使用子查询返回的数据将被用在主查询中作为条件，以进一步限制要检索的数据。

子查询可以与 SELECT、INSERT、UPDATE 和 DELETE 语句一起使用，可伴随着使用运算符如 =、<、>、>=、<=、IN、BETWEEN 等。

以下是子查询必须遵循的几个规则：

- 子查询必须用括号括起来。
- 子查询在 SELECT 子句中只能有一个列，除非在主查询中有多列，与子查询的所选列进行比较。
- ORDER BY 不能用在子查询中，虽然主查询可以使用 ORDER BY。可以在子查询中使用 GROUP BY，功能与 ORDER BY 相同。
- 子查询返回多于一行，只能与多值运算符一起使用，如 IN 运算符。
- BETWEEN 运算符不能与子查询一起使用，但是，BETWEEN 可在子查询内使用。

SELECT 语句中的子查询使用

子查询通常与 SELECT 语句一起使用。基本语法如下：

```
SELECT column_name [, column_name ]
FROM   table1 [, table2 ]
WHERE  column_name OPERATOR
      (SELECT column_name [, column_name ]
      FROM table1 [, table2 ]
      [WHERE])
```

实例

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

现在，让我们检查 SELECT 语句中的子查询使用：

```
sqlite> SELECT *
     FROM COMPANY
     WHERE ID IN (SELECT ID
                  FROM COMPANY
                  WHERE SALARY > 45000) ;
```

这将产生以下结果:

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
```

INSERT 语句中的子查询使用

子查询也可以与 INSERT 语句一起使用。INSERT 语句使用子查询返回的数据插入到另一个表中。在子查询中所选择的数据可以用任何字符、日期或数字函数修改。

基本语法如下：

```
INSERT INTO table_name [ (column1 [, column2 ]) ]
           SELECT [ *|column1 [, column2 ]
           FROM table1 [, table2 ]
           [ WHERE VALUE OPERATOR ]
```

实例

假设 COMPANY_BKP 的结构与 COMPANY 表相似，且可使用相同的 CREATE TABLE 进行创建，只是表名改为 COMPANY_BKP。现在把整个 COMPANY 表复制到 COMPANY_BKP，语法如下：

```
sqlite> INSERT INTO COMPANY_BKP
     SELECT * FROM COMPANY
     WHERE ID IN (SELECT ID
                  FROM COMPANY) ;
```

UPDATE 语句中的子查询使用

子查询可以与 UPDATE 语句结合使用。当通过 UPDATE 语句使用子查询时，表中单个或多个列被更新。

基本语法如下：

```
UPDATE table
SET column_name = new_value
[ WHERE OPERATOR [ VALUE ]
   (SELECT COLUMN_NAME
   FROM TABLE_NAME)
   [ WHERE) ]
```

实例

假设，我们有 COMPANY_BKP 表，是 COMPANY 表的备份。

下面的实例把 COMPANY 表中所有 AGE 大于或等于 27 的客户的 SALARY 更新为原来的 0.50 倍：

```
sqlite> UPDATE COMPANY
     SET SALARY = SALARY * 0.50
     WHERE AGE IN (SELECT AGE FROM COMPANY_BKP
                   WHERE AGE >= 27 );
```

这将影响两行，最后 COMPANY 表中的记录如下：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  10000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       42500.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

DELETE 语句中的子查询使用

子查询可以与 DELETE 语句结合使用，就像上面提到的其他语句一样。

基本语法如下：

```
DELETE FROM TABLE_NAME
[ WHERE OPERATOR [ VALUE ]
   (SELECT COLUMN_NAME
   FROM TABLE_NAME)
   [ WHERE) ]
```

实例

假设，我们有 COMPANY_BKP 表，是 COMPANY 表的备份。

下面的实例删除 COMPANY 表中所有 AGE 大于或等于 27 的客户记录：

```
sqlite> DELETE FROM COMPANY
     WHERE AGE IN (SELECT AGE FROM COMPANY_BKP
                   WHERE AGE > 27 );
```

这将影响两行，最后 COMPANY 表中的记录如下：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       42500.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

SQLite Autoincrement（自动递增）

SQLite 的 **AUTOINCREMENT** 是一个关键字，用于表中的字段值自动递增。我们可以在创建表时在特定的列名称上使用 **AUTOINCREMENT** 关键字实现该字段值的自动增加。

关键字 **AUTOINCREMENT** 只能用于整型（INTEGER）字段。

语法

AUTOINCREMENT 关键字的基本用法如下：

```
CREATE TABLE table_name(
   column1 INTEGER AUTOINCREMENT,
   column2 datatype,
   column3 datatype,
   .....
   columnN datatype,
);
```

实例

假设要创建的 COMPANY 表如下所示：

```
sqlite> CREATE TABLE COMPANY(
   ID INTEGER PRIMARY KEY   AUTOINCREMENT,
   NAME           TEXT      NOT NULL,
   AGE            INT       NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

现在，向 COMPANY 表插入以下记录：

```
INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)
VALUES ( 'Paul', 32, 'California', 20000.00 );

INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)
VALUES ('Allen', 25, 'Texas', 15000.00 );

INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)
VALUES ('Teddy', 23, 'Norway', 20000.00 );

INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)
VALUES ( 'Mark', 25, 'Rich-Mond ', 65000.00 );

INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)
VALUES ( 'David', 27, 'Texas', 85000.00 );

INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)
VALUES ( 'Kim', 22, 'South-Hall', 45000.00 );

INSERT INTO COMPANY (NAME,AGE,ADDRESS,SALARY)
VALUES ( 'James', 24, 'Houston', 10000.00 );
```

这将向 COMPANY 表插入 7 个元组，此时 COMPANY 表的记录如下：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

SQLite 注入

如果您的站点允许用户通过网页输入，并将输入内容插入到 SQLite 数据库中，这个时候您就面临着一个被称为 SQL 注入的安全问题。本章节将向您讲解如何防止这种情况的发生，确保脚本和 SQLite 语句的安全。

注入通常在请求用户输入时发生，比如需要用户输入姓名，但用户却输入了一个 SQLite 语句，而这语句就会在不知不觉中在数据库上运行。

永远不要相信用户提供的数据，所以只处理通过验证的数据，这项规则是通过模式匹配来完成的。在下面的实例中，用户名 username 被限制为字母数字字符或者下划线，长度必须在 8 到 20 个字符之间 - 请根据需要修改这些规则。

```
if (preg_match("/^\w{8,20}$/", $_GET['username'], $matches)){
   $db = new SQLiteDatabase('filename');
   $result = @$db->query("SELECT * FROM users WHERE username=$match
}else{
   echo "username not accepted";
}
```

为了演示这个问题，假设考虑此摘录：To demonstrate the problem, consider this excerpt:

```
$name = "Qadir'; DELETE FROM users;";
@$db->query("SELECT * FROM users WHERE username='{$name}'");
```

函数调用是为了从用户表中检索 name 列与用户指定的名称相匹配的记录。正常情况下，**$name** 只包含字母数字字符或者空格，比如字符串 ilia。但在这里，向 $name 追加了一个全新的查询，这个对数据库的调用将会造成灾难性的问题：注入的 DELETE 查询会删除 users 的所有记录。

虽然已经存在有不允许查询堆叠或在单个函数调用中执行多个查询的数据库接口，如果尝试堆叠查询，则会调用失败，但 SQLite 和 PostgreSQL 里仍进行堆叠查询，即执行在一个字符串中提供的所有查询，这会导致严重的安全问题。

防止 SQL 注入

在脚本语言中，比如 PERL 和 PHP，您可以巧妙地处理所有的转义字符。编程语言 PHP 提供了字符串函数 **sqlite_escape_string()** 来转义对于 SQLite 来说比较特殊的输入字符。

```
if (get_magic_quotes_gpc())
{
  $name = sqlite_escape_string($name);
}
$result = @$db->query("SELECT * FROM users WHERE username='{$name}
```

虽然编码使得插入数据变得安全，但是它会呈现简单的文本比较，在查询中，对于包含二进制数据的列，**LIKE** 子句是不可用的。

请注意，addslashes() 不应该被用在 SQLite 查询中引用字符串，它会在检索数据时导致奇怪的结果。

SQLite Explain（解释）

在 SQLite 语句之前，可以使用 "EXPLAIN" 关键字或 "EXPLAIN QUERY PLAN" 短语，用于描述表的细节。

如果省略了 EXPLAIN 关键字或短语，任何的修改都会引起 SQLite 语句的查询行为，并返回有关 SQLite 语句如何操作的信息。

- 来自 EXPLAIN 和 EXPLAIN QUERY PLAN 的输出只用于交互式分析和排除故障。
- 输出格式的细节可能会随着 SQLite 版本的不同而有所变化。
- 应用程序不应该使用 EXPLAIN 或 EXPLAIN QUERY PLAN，因为其确切的行为是可变的且只有部分会被记录。

语法

EXPLAIN 的语法如下：

```
EXPLAIN [SQLite Query]
```

EXPLAIN QUERY PLAN 的语法如下：

```
EXPLAIN  QUERY PLAN [SQLite Query]
```

实例

假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

现在，让我们检查 SELECT 语句中的 **Explain** 使用：

```
sqlite> EXPLAIN SELECT *  FROM COMPANY  WHERE Salary &gt= 20000;
```

这将产生以下结果：

```
addr        opcode      p1          p2          p3
----------  ----------  ----------  ----------  ----------
0           Goto        0           19
1           Integer     0           0
2           OpenRead    0           8
3           SetNumColu  0           5
4           Rewind      0           17
5           Column      0           4
6           RealAffini  0           0
7           Integer     20000       0
8           Lt          357         16          collseq(BI
9           Rowid       0           0
10          Column      0           1
11          Column      0           2
12          Column      0           3
13          Column      0           4
14          RealAffini  0           0
15          Callback    5           0
16          Next        0           5
17          Close       0           0
18          Halt        0           0
19          Transactio  0           0
20          VerifyCook  0           38
21          Goto        0           1
22          Noop        0           0
```

现在，让我们检查 SELECT 语句中的 **Explain Query Plan** 使用：

```
SQLite> EXPLAIN QUERY PLAN SELECT * FROM COMPANY WHERE Salary &gt=
```

```
order       from        detail
----------  ----------  -------------
0           0           TABLE COMPANY
```

SQLite Vacuum

VACUUM 命令通过复制主数据库中的内容到一个临时数据库文件，然后清空主数据库，并从副本中重新载入原始的数据库文件。这消除了空闲页，把表中的数据排列为连续的，另外会清理数据库文件结构。

如果表中没有明确的整型主键（INTEGER PRIMARY KEY），VACUUM 命令可能会改变表中条目的行 ID（ROWID）。VACUUM 命令只适用于主数据库，附加的数据库文件是不可能使用 VACUUM 命令。

如果有一个活动的事务，VACUUM 命令就会失败。VACUUM 命令是一个用于内存数据库的任何操作。由于 VACUUM 命令从头开始重新创建数据库文件，所以 VACUUM 也可以用于修改许多数据库特定的配置参数。

手动 VACUUM

下面是在命令提示符中对整个数据库发出 VACUUM 命令的语法：

```
$sqlite3 database_name "VACUUM;"
```

您也可以在 SQLite 提示符中运行 VACUUM，如下所示：

```
sqlite> VACUUM;
```

您也可以在特定的表上运行 VACUUM，如下所示：

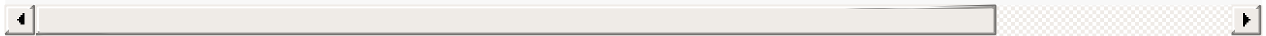
```
sqlite> VACUUM table_name;
```

自动 VACCUM（Auto-VACUUM）

SQLite 的 Auto-VACUUM 与 VACUUM 不大一样，它只是把空闲页移到数据库末尾，从而减小数据库大小。通过这样做，它可以明显地把数据库碎片化，而 VACUUM 则是反碎片化。所以 Auto-VACUUM 只会让数据库更小。

在 SQLite 提示符中，您可以通过下面的编译运行，启用/禁用 SQLite 的 Auto-VACUUM：

```
sqlite> PRAGMA auto_vacuum = NONE;  -- 0 means disable auto vacuum
sqlite> PRAGMA auto_vacuum = INCREMENTAL;  -- 1 means enable increm
sqlite> PRAGMA auto_vacuum = FULL;  -- 2 means enable full auto vac
```



您可以从命令提示符中运行下面的命令来检查 auto-vacuum 设置：

```
$sqlite3 database_name "PRAGMA auto_vacuum;"
```

SQLite 日期 & 时间

SQLite 支持以下五个日期和时间函数：

函数	实例
date(timestring, modifiers...)	以 YYYY-MM-DD 格式返回日期。
time(timestring, modifiers...)	以 HH:MM:SS 格式返回时间。
datetime(timestring, modifiers...)	以 YYYY-MM-DD HH:MM:SS 格式返回。
julianday(timestring, modifiers...)	这将返回从格林尼治时间的公元前 4714 年 11 月 24 日正午算起的天数。
strftime(timestring, modifiers...)	这将根据第一个参数指定的格式字符串返回格式化的日期。具体格式见下边讲解。

上述五个日期和时间函数把时间字符串作为参数。时间字符串后跟零个或多个 modifiers 修饰符。strftime() 函数也可以把格式字符串作为其第一个参数。下面将为您详细讲解不同类型的时间字符串和修饰符。

时间字符串

一个时间字符串可以采用下面任何一种格式：

时间字符串	实例
YYYY-MM-DD	2010-12-30
YYYY-MM-DD HH:MM	2010-12-30 12:10
YYYY-MM-DD HH:MM:SS.SSS	2010-12-30 12:10:04.100
MM-DD-YYYY HH:MM	30-12-2010 12:10
HH:MM	12:10
YYYY-MM-DD**T**HH:MM	2010-12-30 12:10
HH:MM:SS	12:10:01
YYYYMMDD HHMMSS	20101230 121001
now	2013-05-07

您可以使用 "T" 作为分隔日期和时间的文字字符。

修饰符（Modifiers）

时间字符串后边可跟着零个或多个的修饰符，这将改变有上述五个函数返回的日期和/或时间。任何上述五大功能返回时间。修饰符应从左到右使用，下面列出了可在 SQLite 中使用的修饰符：

- NNN days
- NNN hours
- NNN minutes
- NNN.NNNN seconds
- NNN months
- NNN years
- start of month
- start of year
- start of day
- weekday N
- unixepoch
- localtime
- utc

格式化

SQLite 提供了非常方便的函数 **strftime()** 来格式化任何日期和时间。您可以使用以下的替换来格式化日期和时间：

替换	描述
%d	一月中的第几天，01-31
%f	带小数部分的秒，SS.SSS
%H	小时，00-23
%j	一年中的第几天，001-366
%J	儒略日数，DDDD.DDDD
%m	月，00-12
%M	分，00-59
%s	从 1970-01-01 算起的秒数
%S	秒，00-59
%w	一周中的第几天，0-6 (0 is Sunday)
%W	一年中的第几周，01-53
%Y	年，YYYY
%%	% symbol

实例

现在让我们使用 SQLite 提示符尝试不同的实例。下面是计算当前日期：

```
sqlite> SELECT date('now');
2013-05-07
```

下面是计算当前月份的最后一天：

```
sqlite> SELECT date('now','start of month','+1 month','-1 day');
2013-05-31
```

下面是计算给定 UNIX 时间戳 1092941466 的日期和时间：

```
sqlite> SELECT datetime(1092941466, 'unixepoch');
2004-08-19 18:51:06
```

下面是计算给定 UNIX 时间戳 1092941466 相对本地时区的日期和时间：

```
sqlite> SELECT datetime(1092941466, 'unixepoch', 'localtime');
2004-08-19 11:51:06
```

下面是计算当前的 UNIX 时间戳：

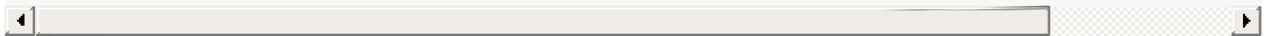
```
sqlite> SELECT datetime(1092941466, 'unixepoch', 'localtime');
1367926057
```

下面是计算美国"独立宣言"签署以来的天数：

```
sqlite> SELECT julianday('now') - julianday('1776-07-04');
86504.4775830326
```

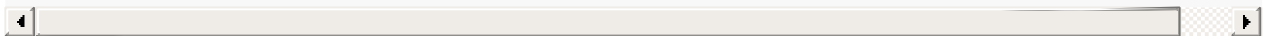
下面是计算从 2004 年某一特定时刻以来的秒数：

```
sqlite> SELECT strftime('%s','now') - strftime('%s','2004-01-01 02
295001572
```



下面是计算当年 10 月的第一个星期二的日期：

```
sqlite> SELECT date('now','start of year','+9 months','weekday 2')
2013-10-01
```



下面是计算从 UNIX 纪元算起的以秒为单位的时间（类似 strftime('%s','now')，不同的是这里有包括小数部分）：

```
sqlite> SELECT (julianday('now') - 2440587.5)*86400.0;
1367926077.12598
```

在 UTC 与本地时间值之间进行转换，当格式化日期时，使用 utc 或 localtime 修饰符，如下所示：

```
sqlite> SELECT time('12:00', 'localtime');
05:00:00
```

```
sqlite>  SELECT time('12:00', 'utc');
19:00:00
```

SQLite 常用函数

SQLite 有许多内置函数用于处理字符串或数字数据。下面列出了一些有用的 SQLite 内置函数，且所有函数都是大小写不敏感，这意味着您可以使用这些函数的小写形式或大写形式或混合形式。欲了解更多详情，请查看 SQLite 的官方文档：

函数	描述
SQLite COUNT 函数	SQLite COUNT 聚集函数是用来计算一个数据库表中的行数。
SQLite MAX 函数	SQLite MAX 聚合函数允许我们选择某列的最大值。
SQLite MIN 函数	SQLite MIN 聚合函数允许我们选择某列的最小值。
SQLite AVG 函数	SQLite AVG 聚合函数计算某列的平均值。
SQLite SUM 函数	SQLite SUM 聚合函数允许为一个数值列计算总和。
SQLite RANDOM 函数	SQLite RANDOM 函数返回一个介于 -9223372036854775808 和 +9223372036854775807 之间的伪随机整数。
SQLite ABS 函数	SQLite ABS 函数返回数值参数的绝对值。
SQLite UPPER 函数	SQLite UPPER 函数把字符串转换为大写字母。
SQLite LOWER 函数	SQLite LOWER 函数把字符串转换为小写字母。
SQLite LENGTH 函数	SQLite LENGTH 函数返回字符串的长度。
SQLite sqlite_version 函数	SQLite sqlite_version 函数返回 SQLite 库的版本。

在我们开始讲解这些函数实例之前，先假设 COMPANY 表有以下记录：

```
ID          NAME        AGE         ADDRESS     SALARY
----------  ----------  ----------  ----------  ----------
1           Paul        32          California  20000.0
2           Allen       25          Texas       15000.0
3           Teddy       23          Norway      20000.0
4           Mark        25          Rich-Mond   65000.0
5           David       27          Texas       85000.0
6           Kim         22          South-Hall  45000.0
7           James       24          Houston     10000.0
```

SQLite COUNT 函数

SQLite COUNT 聚集函数是用来计算一个数据库表中的行数。下面是实例：

```
sqlite> SELECT count(*) FROM COMPANY;
```

上面的 SQLite SQL 语句将产生以下结果：

```
count(*)
----------
7
```

SQLite MAX 函数

SQLite MAX 聚合函数允许我们选择某列的最大值。下面是实例：

```
sqlite> SELECT max(salary) FROM COMPANY;
```

上面的 SQLite SQL 语句将产生以下结果：

```
max(salary)
-----------
85000.0
```

SQLite MIN 函数

SQLite MIN 聚合函数允许我们选择某列的最小值。下面是实例：

```
sqlite> SELECT min(salary) FROM COMPANY;
```

上面的 SQLite SQL 语句将产生以下结果：

```
min(salary)
-----------
10000.0
```

SQLite AVG 函数

SQLite AVG 聚合函数计算某列的平均值。下面是实例：

```
sqlite> SELECT avg(salary) FROM COMPANY;
```

上面的 SQLite SQL 语句将产生以下结果：

```
avg(salary)
----------------
37142.8571428572
```

SQLite SUM 函数

SQLite SUM 聚合函数允许为一个数值列计算总和。下面是实例：

```
sqlite> SELECT sum(salary) FROM COMPANY;
```

上面的 SQLite SQL 语句将产生以下结果：

```
sum(salary)
-----------
260000.0
```

SQLite RANDOM 函数

SQLite RANDOM 函数返回一个介于 -9223372036854775808 和 +9223372036854775807 之间的伪随机整数。下面是实例：

```
sqlite> SELECT random() AS Random;
```

上面的 SQLite SQL 语句将产生以下结果：

```
Random
--------------------
5876796417670984050
```

SQLite ABS 函数

SQLite ABS 函数返回数值参数的绝对值。下面是实例：

```
sqlite> SELECT abs(5), abs(-15), abs(NULL), abs(0), abs("ABC");
```

上面的 SQLite SQL 语句将产生以下结果：

```
abs(5)      abs(-15)    abs(NULL)   abs(0)      abs("ABC")
----------  ----------  ----------  ----------  ----------
5           15                      0           0.0
```

SQLite UPPER 函数

SQLite UPPER 函数把字符串转换为大写字母。下面是实例：

```
sqlite> SELECT upper(name) FROM COMPANY;
```

上面的 SQLite SQL 语句将产生以下结果：

```
upper(name)
-----------
PAUL
ALLEN
TEDDY
MARK
DAVID
KIM
JAMES
```

SQLite LOWER 函数

SQLite LOWER 函数把字符串转换为小写字母。下面是实例：

```
sqlite> SELECT lower(name) FROM COMPANY;
```

上面的 SQLite SQL 语句将产生以下结果：

```
lower(name)
-----------
paul
allen
teddy
mark
david
kim
james
```

SQLite LENGTH 函数

SQLite LENGTH 函数返回字符串的长度。下面是实例：

```
sqlite> SELECT name, length(name) FROM COMPANY;
```

上面的 SQLite SQL 语句将产生以下结果：

```
NAME        length(name)
----------  ------------
Paul        4
Allen       5
Teddy       5
Mark        4
David       5
Kim         3
James       5
```

SQLite sqlite_version 函数

SQLite sqlite_version 函数返回 SQLite 库的版本。下面是实例：

```
sqlite> SELECT sqlite_version() AS 'SQLite Version';
```

上面的 SQLite SQL 语句将产生以下结果：

```
SQLite Version
--------------
3.6.20
```

SQLite 接口

SQLite - C/C++

安装

在 C/C++ 程序中使用 SQLite 之前，我们需要确保机器上已经有 SQLite 库。可以查看 SQLite 安装章节了解安装过程。

C/C++ 接口 API

以下是重要的 C&C++ / SQLite 接口程序，可以满足您在 C/C++ 程序中使用 SQLite 数据库的需求。如果您需要了解更多细节，请查看 SQLite 官方文档。

API	描述
sqlite3_open(const char *filename, sqlite3 **ppDb)	该例程打开一个指向 SQLite 数据库文件的连接，返回一个用于其他 SQLite 程序的数据库连接对象。如果 *filename* 参数是 NULL 或 ':memory:'，那么 sqlite3_open() 将会在 RAM 中创建一个内存数据库，这只会在 session 的有效时间内持续。 如果文件名 filename 不为 NULL，那么 sqlite3_open() 将使用这个参数值尝试打开数据库文件。如果该名称的文件不存在，sqlite3_open() 将创建一个新的命名为该名称的数据库文件并打开。
sqlite3_exec(sqlite3*, const char *sql, sqlite_callback, void *data, char **errmsg)	该例程提供了一个执行 SQL 命令的快捷方式，SQL 命令由 sql 参数提供，可以由多个 SQL 命令组成。在这里，第一个参数 *sqlite3* 是打开的数据库对象，*sqlite_callback* 是一个回调，*data* 作为其第一个参数，errmsg 将被返回用来获取程序生成的任何错误。 sqlite3_exec() 程序解析并执行由 **sql** 参数所给的每个命令，直到字符串结束或者遇到错误为止。
sqlite3_close(sqlite3*)	该例程关闭之前调用 sqlite3_open() 打开的数据库连接。所有与连接相关的语句都应在连接关闭之前完成。 如果还有查询没有完成，sqlite3_close() 将返回 SQLITE_BUSY 禁止关闭的错误消息。

连接数据库

下面的 C 代码段显示了如何连接到一个现有的数据库。如果数据库不存在，那么它就会被创建，最后将返回一个数据库对象。

```
#include <stdio.h>
#include <sqlite3.h>

int main(int argc, char* argv[])
{
   sqlite3 *db;
   char *zErrMsg = 0;
   int rc;

   rc = sqlite3_open("test.db", &db);

   if( rc ){
      fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(d
      exit(0);
   }else{
      fprintf(stderr, "Opened database successfully\n");
   }
   sqlite3_close(db);
}
```

现在，让我们来编译和运行上面的程序，在当前目录中创建我们的数据库 **test.db**。您可以根据需要改变路径。

```
$gcc test.c -l sqlite3
$./a.out
Opened database successfully
```

如果要使用 C++ 源代码，可以按照下列所示编译代码：

```
$g++ test.c -l sqlite3
```

在这里，把我们的程序链接上 sqlite3 库，以便向 C 程序提供必要的函数。这将在您的目录下创建一个数据库文件 test.db，您将得到如下结果：

```
-rwxr-xr-x. 1 root root 7383 May  8 02:06 a.out
-rw-r--r--. 1 root root  323 May  8 02:05 test.c
-rw-r--r--. 1 root root    0 May  8 02:06 test.db
```

创建表

下面的 C 代码段将用于在先前创建的数据库中创建一个表：

```
#include <stdio.h>
#include <stdlib.h>
#include <sqlite3.h>

static int callback(void *NotUsed, int argc, char **argv, char **az
   int i;
   for(i=0; i&lt;argc; i++){
      printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL")
   }
   printf("\n");
   return 0;
}

int main(int argc, char* argv[])
{
   sqlite3 *db;
   char *zErrMsg = 0;
   int  rc;
   char *sql;

   /* Open database */
   rc = sqlite3_open("test.db", &db);
   if( rc ){
      fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(d
      exit(0);
   }else{
      fprintf(stdout, "Opened database successfully\n");
   }

   /* Create SQL statement */
   sql = "CREATE TABLE COMPANY("  \
         "ID INT PRIMARY KEY     NOT NULL," \
         "NAME           TEXT    NOT NULL," \
         "AGE            INT     NOT NULL," \
         "ADDRESS        CHAR(50)," \
         "SALARY         REAL );";

   /* Execute SQL statement */
   rc = sqlite3_exec(db, sql, callback, 0, &zErrMsg);
   if( rc != SQLITE_OK ){
   fprintf(stderr, "SQL error: %s\n", zErrMsg);
      sqlite3_free(zErrMsg);
   }else{
      fprintf(stdout, "Table created successfully\n");
   }
   sqlite3_close(db);
   return 0;
}
```

上述程序编译和执行时，它会在 test.db 文件中创建 COMPANY 表，最终文件列表如下所示：

```
-rwxr-xr-x. 1 root root 9567 May  8 02:31 a.out
-rw-r--r--. 1 root root 1207 May  8 02:31 test.c
-rw-r--r--. 1 root root 3072 May  8 02:31 test.db
```

INSERT 操作

下面的 C 代码段显示了如何在上面创建的 COMPANY 表中创建记录：

```
#include <stdio.h>
#include <stdlib.h>
#include <sqlite3.h>

static int callback(void *NotUsed, int argc, char **argv, char **az
   int i;
   for(i=0; i&lt;argc; i++){
      printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL")
   }
   printf("\n");
   return 0;
}

int main(int argc, char* argv[])
{
   sqlite3 *db;
   char *zErrMsg = 0;
   int rc;
   char *sql;

   /* Open database */
   rc = sqlite3_open("test.db", &db);
   if( rc ){
      fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(d
      exit(0);
   }else{
      fprintf(stderr, "Opened database successfully\n");
   }

   /* Create SQL statement */
   sql = "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) "  \
         "VALUES (1, 'Paul', 32, 'California', 20000.00 ); " \
         "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) "  \
         "VALUES (2, 'Allen', 25, 'Texas', 15000.00 ); "     \
         "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)" \
         "VALUES (3, 'Teddy', 23, 'Norway', 20000.00 );" \
         "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)" \
         "VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 );";

   /* Execute SQL statement */
   rc = sqlite3_exec(db, sql, callback, 0, &zErrMsg);
   if( rc != SQLITE_OK ){
      fprintf(stderr, "SQL error: %s\n", zErrMsg);
      sqlite3_free(zErrMsg);
   }else{
      fprintf(stdout, "Records created successfully\n");
   }
   sqlite3_close(db);
   return 0;
}
```

上述程序编译和执行时，它会在 COMPANY 表中创建给定记录，并会显示以下两行：

```
Opened database successfully
Records created successfully
```

SELECT 操作

在我们开始讲解获取记录的实例之前，让我们先了解下回调函数的一些细节，这将在我们的实例使用到。这个回调提供了一个从 SELECT 语句获得结果的方式。它声明如下：

```
typedef int (*sqlite3_callback)(
void*,    /* Data provided in the 4th argument of sqlite3_exec() */
int,      /* The number of columns in row */
char**,   /* An array of strings representing fields in the row */
char**    /* An array of strings representing column names */
);
```

如果上面的回调在 sqlite_exec() 程序中作为第三个参数，那么 SQLite 将为 SQL 参数内执行的每个 SELECT 语句中处理的每个记录调用这个回调函数。

下面的 C 代码段显示了如何从前面创建的 COMPANY 表中获取并显示记录：

```
#include <stdio.h>
#include <stdlib.h>
#include <sqlite3.h>

static int callback(void *data, int argc, char **argv, char **azCol
   int i;
   fprintf(stderr, "%s: ", (const char*)data);
   for(i=0; i&lt;argc; i++){
      printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL")
   }
   printf("\n");
   return 0;
}

int main(int argc, char* argv[])
{
   sqlite3 *db;
   char *zErrMsg = 0;
   int rc;
   char *sql;
   const char* data = "Callback function called";

   /* Open database */
   rc = sqlite3_open("test.db", &db);
   if( rc ){
      fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(d
      exit(0);
   }else{
      fprintf(stderr, "Opened database successfully\n");
   }

   /* Create SQL statement */
   sql = "SELECT * from COMPANY";

   /* Execute SQL statement */
   rc = sqlite3_exec(db, sql, callback, (void*)data, &zErrMsg);
   if( rc != SQLITE_OK ){
      fprintf(stderr, "SQL error: %s\n", zErrMsg);
      sqlite3_free(zErrMsg);
   }else{
      fprintf(stdout, "Operation done successfully\n");
   }
   sqlite3_close(db);
   return 0;
}
```

上述程序编译和执行时，它会产生以下结果：

```
Opened database successfully
Callback function called: ID = 1
NAME = Paul
AGE = 32
ADDRESS = California
SALARY = 20000.0

Callback function called: ID = 2
NAME = Allen
AGE = 25
ADDRESS = Texas
SALARY = 15000.0

Callback function called: ID = 3
NAME = Teddy
AGE = 23
ADDRESS = Norway
SALARY = 20000.0

Callback function called: ID = 4
NAME = Mark
AGE = 25
ADDRESS = Rich-Mond
SALARY = 65000.0

Operation done successfully
```

UPDATE 操作

下面的 C 代码段显示了如何使用 UPDATE 语句来更新任何记录，然后从 COMPANY 表中获取并显示更新的记录：

```
#include <stdio.h>
#include <stdlib.h>
#include <sqlite3.h>

static int callback(void *data, int argc, char **argv, char **azCol
   int i;
   fprintf(stderr, "%s: ", (const char*)data);
   for(i=0; i&lt;argc; i++){
      printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL")
   }
   printf("\n");
   return 0;
}

int main(int argc, char* argv[])
{
   sqlite3 *db;
   char *zErrMsg = 0;
   int rc;
   char *sql;
   const char* data = "Callback function called";

   /* Open database */
   rc = sqlite3_open("test.db", &db);
   if( rc ){
      fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(d
      exit(0);
   }else{
      fprintf(stderr, "Opened database successfully\n");
   }

   /* Create merged SQL statement */
   sql = "UPDATE COMPANY set SALARY = 25000.00 where ID=1; " \
         "SELECT * from COMPANY";

   /* Execute SQL statement */
   rc = sqlite3_exec(db, sql, callback, (void*)data, &zErrMsg);
   if( rc != SQLITE_OK ){
      fprintf(stderr, "SQL error: %s\n", zErrMsg);
      sqlite3_free(zErrMsg);
   }else{
      fprintf(stdout, "Operation done successfully\n");
   }
   sqlite3_close(db);
   return 0;
}
```

上述程序编译和执行时，它会产生以下结果：

```
Opened database successfully
Callback function called: ID = 1
NAME = Paul
AGE = 32
ADDRESS = California
SALARY = 25000.0

Callback function called: ID = 2
NAME = Allen
AGE = 25
ADDRESS = Texas
SALARY = 15000.0

Callback function called: ID = 3
NAME = Teddy
AGE = 23
ADDRESS = Norway
SALARY = 20000.0

Callback function called: ID = 4
NAME = Mark
AGE = 25
ADDRESS = Rich-Mond
SALARY = 65000.0

Operation done successfully
```

DELETE 操作

下面的 C 代码段显示了如何使用 DELETE 语句删除任何记录，然后从 COMPANY 表中获取并显示剩余的记录：

```
#include <stdio.h>
#include <stdlib.h>
#include <sqlite3.h>

static int callback(void *data, int argc, char **argv, char **azCol
   int i;
   fprintf(stderr, "%s: ", (const char*)data);
   for(i=0; i<argc; i++){
      printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL")
   }
   printf("\n");
   return 0;
}

int main(int argc, char* argv[])
{
   sqlite3 *db;
   char *zErrMsg = 0;
   int rc;
   char *sql;
   const char* data = "Callback function called";

   /* Open database */
   rc = sqlite3_open("test.db", &db);
   if( rc ){
      fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(d
      exit(0);
   }else{
      fprintf(stderr, "Opened database successfully\n");
   }

   /* Create merged SQL statement */
   sql = "DELETE from COMPANY where ID=2; " \
         "SELECT * from COMPANY";

   /* Execute SQL statement */
   rc = sqlite3_exec(db, sql, callback, (void*)data, &zErrMsg);
   if( rc != SQLITE_OK ){
      fprintf(stderr, "SQL error: %s\n", zErrMsg);
      sqlite3_free(zErrMsg);
   }else{
      fprintf(stdout, "Operation done successfully\n");
   }
   sqlite3_close(db);
   return 0;
}
```

上述程序编译和执行时，它会产生以下结果：

```
Opened database successfully
Callback function called: ID = 1
NAME = Paul
AGE = 32
ADDRESS = California
SALARY = 20000.0

Callback function called: ID = 3
NAME = Teddy
AGE = 23
ADDRESS = Norway
SALARY = 20000.0

Callback function called: ID = 4
NAME = Mark
AGE = 25
ADDRESS = Rich-Mond
SALARY = 65000.0

Operation done successfully
```

SQLite - Java

安装

在 Java 程序中使用 SQLite 之前，我们需要确保机器上已经有 SQLite JDBC Driver 驱动程序和 Java。可以查看 Java 教程了解如何在计算机上安装 Java。现在，我们来看看如何在机器上安装 SQLite JDBC 驱动程序。

- 从 sqlite-jdbc 库下载 *sqlite-jdbc-(VERSION).jar* 的最新版本。
- 在您的 class 路径中添加下载的 jar 文件 *sqlite-jdbc-(VERSION).jar*，或者在 -classpath 选项中使用它，这将在后面的实例中进行讲解。

在学习下面部分的知识之前，您必须对 Java JDBC 概念有初步了解。如果您还未了解相关知识，那么建议您可以先花半个小时学习下 JDBC 教程相关知识，这将有助于您学习接下来讲解的知识。

连接数据库

下面的 Java 程序显示了如何连接到一个现有的数据库。如果数据库不存在，那么它就会被创建，最后将返回一个数据库对象。

```
import java.sql.*;

public class SQLiteJDBC
{
  public static void main( String args[] )
  {
    Connection c = null;
    try {
      Class.forName("org.sqlite.JDBC");
      c = DriverManager.getConnection("jdbc:sqlite:test.db");
    } catch ( Exception e ) {
      System.err.println( e.getClass().getName() + ": " + e.getMess
      System.exit(0);
    }
    System.out.println("Opened database successfully");
  }
}
```

现在，让我们来编译和运行上面的程序，在当前目录中创建我们的数据库 **test.db**。您可以根据需要改变路径。我们假设当前路径下可用的 JDBC 驱动程序的版本是 *sqlite-jdbc-3.7.2.jar*。

```
$javac SQLiteJDBC.java
$java -classpath ".:sqlite-jdbc-3.7.2.jar" SQLiteJDBC
Open database successfully
```

如果您想要使用 Windows 机器，可以按照下列所示编译和运行您的代码：

```
$javac SQLiteJDBC.java
$java -classpath ".;sqlite-jdbc-3.7.2.jar" SQLiteJDBC
Opened database successfully
```

创建表

下面的 Java 程序将用于在先前创建的数据库中创建一个表：

```
import java.sql.*;

public class SQLiteJDBC
{
  public static void main( String args[] )
  {
    Connection c = null;
    Statement stmt = null;
    try {
      Class.forName("org.sqlite.JDBC");
      c = DriverManager.getConnection("jdbc:sqlite:test.db");
      System.out.println("Opened database successfully");

      stmt = c.createStatement();
      String sql = "CREATE TABLE COMPANY " +
                   "(ID INT PRIMARY KEY     NOT NULL," +
                   " NAME           TEXT    NOT NULL, " +
                   " AGE            INT     NOT NULL, " +
                   " ADDRESS        CHAR(50), " +
                   " SALARY         REAL)";
      stmt.executeUpdate(sql);
      stmt.close();
      c.close();
    } catch ( Exception e ) {
      System.err.println( e.getClass().getName() + ": " + e.getMess
      System.exit(0);
    }
    System.out.println("Table created successfully");
  }
}
```

上述程序编译和执行时，它会在 **test.db** 中创建 COMPANY 表，最终文件列表如下所示：

```
-rw-r--r--. 1 root root 3201128 Jan 22 19:04 sqlite-jdbc-3.7.2.jar
-rw-r--r--. 1 root root    1506 May  8 05:43 SQLiteJDBC.class
-rw-r--r--. 1 root root     832 May  8 05:42 SQLiteJDBC.java
-rw-r--r--. 1 root root    3072 May  8 05:43 test.db
```

INSERT 操作

下面的 Java 代码显示了如何在上面创建的 COMPANY 表中创建记录：

```
import java.sql.*;

public class SQLiteJDBC
{
  public static void main( String args[] )
  {
    Connection c = null;
    Statement stmt = null;
    try {
      Class.forName("org.sqlite.JDBC");
      c = DriverManager.getConnection("jdbc:sqlite:test.db");
      c.setAutoCommit(false);
      System.out.println("Opened database successfully");

      stmt = c.createStatement();
      String sql = "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY
                   "VALUES (1, 'Paul', 32, 'California', 20000.00 
      stmt.executeUpdate(sql);

      sql = "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) " +
            "VALUES (2, 'Allen', 25, 'Texas', 15000.00 );";
      stmt.executeUpdate(sql);

      sql = "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) " +
            "VALUES (3, 'Teddy', 23, 'Norway', 20000.00 );";
      stmt.executeUpdate(sql);

      sql = "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) " +
            "VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 );";
      stmt.executeUpdate(sql);

      stmt.close();
      c.commit();
      c.close();
    } catch ( Exception e ) {
      System.err.println( e.getClass().getName() + ": " + e.getMess
      System.exit(0);
    }
    System.out.println("Records created successfully");
  }
}
```

上述程序编译和执行时，它会在 COMPANY 表中创建给定记录，并会显示以下两行：

```
Opened database successfully
Records created successfully
```

SELECT 操作

下面的 Java 程序显示了如何从前面创建的 COMPANY 表中获取并显示记录：

```
import java.sql.*;

public class SQLiteJDBC
{
  public static void main( String args[] )
  {
    Connection c = null;
    Statement stmt = null;
    try {
      Class.forName("org.sqlite.JDBC");
      c = DriverManager.getConnection("jdbc:sqlite:test.db");
      c.setAutoCommit(false);
      System.out.println("Opened database successfully");

      stmt = c.createStatement();
      ResultSet rs = stmt.executeQuery( "SELECT * FROM COMPANY;" );
      while ( rs.next() ) {
         int id = rs.getInt("id");
         String  name = rs.getString("name");
         int age  = rs.getInt("age");
         String  address = rs.getString("address");
         float salary = rs.getFloat("salary");
         System.out.println( "ID = " + id );
         System.out.println( "NAME = " + name );
         System.out.println( "AGE = " + age );
         System.out.println( "ADDRESS = " + address );
         System.out.println( "SALARY = " + salary );
         System.out.println();
      }
      rs.close();
      stmt.close();
      c.close();
    } catch ( Exception e ) {
      System.err.println( e.getClass().getName() + ": " + e.getMess
      System.exit(0);
    }
    System.out.println("Operation done successfully");
  }
}
```

上述程序编译和执行时，它会产生以下结果：

```
Opened database successfully
ID = 1
NAME = Paul
AGE = 32
ADDRESS = California
SALARY = 20000.0

ID = 2
NAME = Allen
AGE = 25
ADDRESS = Texas
SALARY = 15000.0

ID = 3
NAME = Teddy
AGE = 23
ADDRESS = Norway
SALARY = 20000.0

ID = 4
NAME = Mark
AGE = 25
ADDRESS = Rich-Mond
SALARY = 65000.0

Operation done successfully
```

UPDATE 操作

下面的 Java 代码显示了如何使用 UPDATE 语句来更新任何记录，然后从 COMPANY 表中获取并显示更新的记录：

```
import java.sql.*;

public class SQLiteJDBC
{
  public static void main( String args[] )
  {
    Connection c = null;
    Statement stmt = null;
    try {
      Class.forName("org.sqlite.JDBC");
      c = DriverManager.getConnection("jdbc:sqlite:test.db");
      c.setAutoCommit(false);
      System.out.println("Opened database successfully");

      stmt = c.createStatement();
      String sql = "UPDATE COMPANY set SALARY = 25000.00 where ID=1
      stmt.executeUpdate(sql);
      c.commit();

      ResultSet rs = stmt.executeQuery( "SELECT * FROM COMPANY;" );
      while ( rs.next() ) {
         int id = rs.getInt("id");
         String  name = rs.getString("name");
         int age  = rs.getInt("age");
         String  address = rs.getString("address");
         float salary = rs.getFloat("salary");
         System.out.println( "ID = " + id );
         System.out.println( "NAME = " + name );
         System.out.println( "AGE = " + age );
         System.out.println( "ADDRESS = " + address );
         System.out.println( "SALARY = " + salary );
         System.out.println();
      }
      rs.close();
      stmt.close();
      c.close();
    } catch ( Exception e ) {
      System.err.println( e.getClass().getName() + ": " + e.getMess
      System.exit(0);
    }
    System.out.println("Operation done successfully");
  }
}
```

上述程序编译和执行时，它会产生以下结果：

```
Opened database successfully
ID = 1
NAME = Paul
AGE = 32
ADDRESS = California
SALARY = 25000.0

ID = 2
NAME = Allen
AGE = 25
ADDRESS = Texas
SALARY = 15000.0

ID = 3
NAME = Teddy
AGE = 23
ADDRESS = Norway
SALARY = 20000.0

ID = 4
NAME = Mark
AGE = 25
ADDRESS = Rich-Mond
SALARY = 65000.0

Operation done successfully
```

DELETE 操作

下面的 Java 代码显示了如何使用 DELETE 语句删除任何记录，然后从 COMPANY 表中获取并显示剩余的记录：

```
import java.sql.*;

public class SQLiteJDBC
{
  public static void main( String args[] )
  {
    Connection c = null;
    Statement stmt = null;
    try {
      Class.forName("org.sqlite.JDBC");
      c = DriverManager.getConnection("jdbc:sqlite:test.db");
      c.setAutoCommit(false);
      System.out.println("Opened database successfully");

      stmt = c.createStatement();
      String sql = "DELETE from COMPANY where ID=2;";
      stmt.executeUpdate(sql);
      c.commit();

      ResultSet rs = stmt.executeQuery( "SELECT * FROM COMPANY;" );
      while ( rs.next() ) {
         int id = rs.getInt("id");
         String  name = rs.getString("name");
         int age  = rs.getInt("age");
         String  address = rs.getString("address");
         float salary = rs.getFloat("salary");
         System.out.println( "ID = " + id );
         System.out.println( "NAME = " + name );
         System.out.println( "AGE = " + age );
         System.out.println( "ADDRESS = " + address );
         System.out.println( "SALARY = " + salary );
         System.out.println();
      }
      rs.close();
      stmt.close();
      c.close();
    } catch ( Exception e ) {
      System.err.println( e.getClass().getName() + ": " + e.getMess
      System.exit(0);
    }
    System.out.println("Operation done successfully");
  }
}
```

上述程序编译和执行时，它会产生以下结果：

```
Opened database successfully
ID = 1
NAME = Paul
AGE = 32
ADDRESS = California
SALARY = 25000.0

ID = 3
NAME = Teddy
AGE = 23
ADDRESS = Norway
SALARY = 20000.0

ID = 4
NAME = Mark
AGE = 25
ADDRESS = Rich-Mond
SALARY = 65000.0

Operation done successfully
```

SQLite - PHP

安装

自 PHP 5.3.0 起默认启用 SQLite3 扩展。可以在编译时使用 **--without-sqlite3** 禁用 SQLite3 扩展。

Windows 用户必须启用 php_sqlite3.dll 才能使用该扩展。自 PHP 5.3.0 起，这个 DLL 被包含在 PHP 的 Windows 分发版中。

如需了解详细的安装指导，建议查看我们的 PHP 教程和它的官方网站。

PHP 接口 API

以下是重要的 PHP 程序，可以满足您在 PHP 程序中使用 SQLite 数据库的需求。如果您需要了解更多细节，请查看 PHP 官方文档。

API	描述
public void SQLite3::open (filename, flags, encryption_key)	打开一个 SQLite 3 数据库。如果构建包括加密，那么它将尝试使用的密钥。 如果文件名 *filename* 赋值为 **':memory:'**，那么 SQLite3::open() 将会在 RAM 中创建一个内存数据库，这只会在 session 的有效时间内持续。 如果文件名 filename 为实际的设备文件名称，那么 SQLite3::open() 将使用这个参数值尝试打开数据库文件。如果该名称的文件不存在，那么将创建一个新的命名为该名称的数据库文件。 可选的 flags 用于判断是否打开 SQLite 数据库。默认情况下，当使用 SQLITE3_OPEN_READWRITE | SQLITE3_OPEN_CREATE 时打开。
public bool SQLite3::exec (string $query)	该例程提供了一个执行 SQL 命令的快捷方式，SQL 命令由 sql 参数提供，可以由多个 SQL 命令组成。该程序用于对给定的数据库执行一个无结果的查询。
public SQLite3Result SQLite3::query (string $query)	该例程执行一个 SQL 查询，如果查询到返回结果则返回一个 **SQLite3Result** 对象。
public int SQLite3::lastErrorCode (void)	该例程返回最近一次失败的 SQLite 请求的数值结果代码。
public string SQLite3::lastErrorMsg (void)	该例程返回最近一次失败的 SQLite 请求的英语文本描述。
public int SQLite3::changes (void)	该例程返回最近一次的 SQL 语句更新或插入或删除的数据库行数。
public bool SQLite3::close (void)	该例程关闭之前调用 SQLite3::open() 打开的数据库连接。
public string SQLite3::escapeString (string $value)	该例程返回一个字符串，在 SQL 语句中，出于安全考虑，该字符串已被正确地转义。

连接数据库

下面的 PHP 代码显示了如何连接到一个现有的数据库。如果数据库不存在，那么它就会被创建，最后将返回一个数据库对象。

```
<?php
   class MyDB extends SQLite3
   {
      function __construct()
      {
         $this->open('test.db');
      }
   }
   $db = new MyDB();
   if(!$db){
      echo $db->lastErrorMsg();
   } else {
      echo "Opened database successfully\n";
   }
?>
```

现在，让我们来运行上面的程序，在当前目录中创建我们的数据库 **test.db**。您可以根据需要改变路径。如果数据库成功创建，那么会显示下面所示的消息：

```
Open database successfully
```

创建表

下面的 PHP 代码段将用于在先前创建的数据库中创建一个表：

```
<?php
   class MyDB extends SQLite3
   {
      function __construct()
      {
         $this->open('test.db');
      }
   }
   $db = new MyDB();
   if(!$db){
      echo $db->lastErrorMsg();
   } else {
      echo "Opened database successfully\n";
   }

   $sql =<<<EOF
      CREATE TABLE COMPANY
      (ID INT PRIMARY KEY     NOT NULL,
      NAME           TEXT    NOT NULL,
      AGE            INT     NOT NULL,
      ADDRESS        CHAR(50),
      SALARY         REAL);
EOF;

   $ret = $db->exec($sql);
   if(!$ret){
      echo $db->lastErrorMsg();
   } else {
      echo "Table created successfully\n";
   }
   $db->close();
?>
```

上述程序执行时，它会在 **test.db** 中创建 COMPANY 表，并显示下面所示的消息：

```
Opened database successfully
Table created successfully
```

INSERT 操作

下面的 PHP 程序显示了如何在上面创建的 COMPANY 表中创建记录：

```
<?php
   class MyDB extends SQLite3
   {
      function __construct()
      {
         $this->open('test.db');
      }
   }
   $db = new MyDB();
   if(!$db){
      echo $db->lastErrorMsg();
   } else {
      echo "Opened database successfully\n";
   }

   $sql =<<<EOF
      INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
      VALUES (1, 'Paul', 32, 'California', 20000.00 );

      INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
      VALUES (2, 'Allen', 25, 'Texas', 15000.00 );

      INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
      VALUES (3, 'Teddy', 23, 'Norway', 20000.00 );

      INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
      VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 );
EOF;

   $ret = $db->exec($sql);
   if(!$ret){
      echo $db->lastErrorMsg();
   } else {
      echo "Records created successfully\n";
   }
   $db->close();
?>
```

上述程序执行时，它会在 COMPANY 表中创建给定记录，并会显示以下两行：

```
Opened database successfully
Records created successfully
```

SELECT 操作

下面的 PHP 程序显示了如何从前面创建的 COMPANY 表中获取并显示记录：

```
<?php
   class MyDB extends SQLite3
   {
      function __construct()
      {
         $this->open('test.db');
      }
   }
   $db = new MyDB();
   if(!$db){
      echo $db->lastErrorMsg();
   } else {
      echo "Opened database successfully\n";
   }

   $sql =<<<EOF
      SELECT * from COMPANY;
EOF;

   $ret = $db->query($sql);
   while($row = $ret->fetchArray(SQLITE3_ASSOC) ){
      echo "ID = ". $row['ID'] . "\n";
      echo "NAME = ". $row['NAME'] ."\n";
      echo "ADDRESS = ". $row['ADDRESS'] ."\n";
      echo "SALARY =  ".$row['SALARY'] ."\n\n";
   }
   echo "Operation done successfully\n";
   $db->close();
?>
```

上述程序执行时，它会产生以下结果：

```
Opened database successfully
ID = 1
NAME = Paul
ADDRESS = California
SALARY =  20000

ID = 2
NAME = Allen
ADDRESS = Texas
SALARY =  15000

ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY =  20000

ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY =  65000

Operation done successfully
```

UPDATE 操作

下面的 PHP 代码显示了如何使用 UPDATE 语句来更新任何记录，然后从 COMPANY 表中获取并显示更新的记录：

```
<?php
   class MyDB extends SQLite3
   {
      function __construct()
      {
         $this->open('test.db');
      }
   }
   $db = new MyDB();
   if(!$db){
      echo $db->lastErrorMsg();
   } else {
      echo "Opened database successfully\n";
   }
   $sql =<<<EOF
      UPDATE COMPANY set SALARY = 25000.00 where ID=1;
EOF;
   $ret = $db->exec($sql);
   if(!$ret){
      echo $db->lastErrorMsg();
   } else {
      echo $db->changes(), " Record updated successfully\n";
   }

   $sql =<<<EOF
      SELECT * from COMPANY;
EOF;
   $ret = $db->query($sql);
   while($row = $ret->fetchArray(SQLITE3_ASSOC) ){
      echo "ID = ". $row['ID'] . "\n";
      echo "NAME = ". $row['NAME'] ."\n";
      echo "ADDRESS = ". $row['ADDRESS'] ."\n";
      echo "SALARY =  ".$row['SALARY'] ."\n\n";
   }
   echo "Operation done successfully\n";
   $db->close();
?>
```

上述程序执行时，它会产生以下结果：

```
Opened database successfully
1 Record updated successfully
ID = 1
NAME = Paul
ADDRESS = California
SALARY =  25000

ID = 2
NAME = Allen
ADDRESS = Texas
SALARY =  15000

ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY =  20000

ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY =  65000

Operation done successfully
```

DELETE 操作

下面的 PHP 代码显示了如何使用 DELETE 语句删除任何记录，然后从 COMPANY 表中获取并显示剩余的记录：

```
<?php
   class MyDB extends SQLite3
   {
      function __construct()
      {
         $this->open('test.db');
      }
   }
   $db = new MyDB();
   if(!$db){
      echo $db->lastErrorMsg();
   } else {
      echo "Opened database successfully\n";
   }
   $sql =<<<EOF
      DELETE from COMPANY where ID=2;
EOF;
   $ret = $db->exec($sql);
   if(!$ret){
     echo $db->lastErrorMsg();
   } else {
      echo $db->changes(), " Record deleted successfully\n";
   }

   $sql =<<<EOF
      SELECT * from COMPANY;
EOF;
   $ret = $db->query($sql);
   while($row = $ret->fetchArray(SQLITE3_ASSOC) ){
      echo "ID = ". $row['ID'] . "\n";
      echo "NAME = ". $row['NAME'] ."\n";
      echo "ADDRESS = ". $row['ADDRESS'] ."\n";
      echo "SALARY =  ".$row['SALARY'] ."\n\n";
   }
   echo "Operation done successfully\n";
   $db->close();
?>
```

上述程序执行时，它会产生以下结果：

```
Opened database successfully
1 Record deleted successfully
ID = 1
NAME = Paul
ADDRESS = California
SALARY =  25000

ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY =  20000

ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY =  65000

Operation done successfully
```

SQLite - Perl

安装

SQLite3 可使用 Perl DBI 模块与 Perl 进行集成。Perl DBI 模块是 Perl 编程语言的数据库访问模块。它定义了一组提供标准数据库接口的方法、变量及规则。

下面显示了在 Linux/UNIX 机器上安装 DBI 模块的简单步骤：

```
$ wget http://search.cpan.org/CPAN/authors/id/T/TI/TIMB/DBI-1.625.t
$ tar xvfz DBI-1.625.tar.gz
$ cd DBI-1.625
$ perl Makefile.PL
$ make
$ make install
```

如果您需要为 DBI 安装 SQLite 驱动程序，那么可按照以下步骤进行安装：

```
$ wget http://search.cpan.org/CPAN/authors/id/M/MS/MSERGEANT/DBD-SQ
$ tar xvfz DBD-SQLite-1.11.tar.gz
$ cd DBD-SQLite-1.11
$ perl Makefile.PL
$ make
$ make install
```

DBI 接口 API

以下是重要的 DBI 程序，可以满足您在 Perl 程序中使用 SQLite 数据库的需求。如果您需要了解更多细节，请查看 Perl DBI 官方文档。

API	描述
DBI->connect($data_source, "", "", \%attr)	建立一个到被请求的 $data*source* 的数据库连接或者 *session*。如果连接成功，则返回一个数据库处理对象。 数据源形式如下所示：***DBI:SQLite:dbname='test.db'***。其中，*SQLite* 是 *SQLite* 驱动程序名称，*test.db* 是 *SQLite* 数据库文件的名称。如果文件名 *_filename* 赋值为 **':memory:'**，那么它将会在 RAM 中创建一个内存数据库，这只会在 session 的有效时间内持续。 如果文件名 filename 为实际的设备文件名称，那么它将使用这个参数值尝试打开数据库文件。如果该名称的文件不存在，那么将创建一个新的命名为该名称的数据库文件。 您可以保留第二个和第三个参数为空白字符串，最后一个参数用于传递各种属性，详见下面的实例讲解。
$dbh->do($sql)	该例程准备并执行一个简单的 SQL 语句。返回受影响的行数，如果发生错误则返回 undef。返回值 -1 意味着行数未知，或不适用 ，或不可用。在这里，$dbh 是由 DBI->connect() 调用返回的处理。
$dbh->prepare($sql)	该例程为数据库引擎后续执行准备一个语句，并返回一个语句处理对象。
$sth->execute()	该例程执行任何执行预准备的语句需要的处理。如果发生错误则返回 undef。如果成功执行，则无论受影响的行数是多少，总是返回 true。在这里，$sth 是由 $dbh->prepare($sql) 调用返回的语句处理。
$sth->fetchrow_array()	该例程获取下一行数据，并以包含各字段值的列表形式返回。在该列表中，Null 字段将作为 undef 值返回。
$DBI::err	这相当于 $h->err。其中，$h 是任何的处理类型，比如 $dbh、$sth 或 $drh。该程序返回最后调用的驱动程序（driver）方法的数据库引擎错误代码。
$DBI::errstr	这相当于 $h->errstr。其中，$h 是任何的处理类型，比如 $dbh、$sth 或 $drh。该程序返回最后调用的 DBI 方法的数据库引擎错误消息。
$dbh->disconnect()	该例程关闭之前调用 DBI->connect() 打开的数据库连接。

连接数据库

下面的 Perl 代码显示了如何连接到一个现有的数据库。如果数据库不存在，那么它就会被创建，最后将返回一个数据库对象。

```
#!/usr/bin/perl

use DBI;
use strict;

my $driver   = "SQLite";
my $database = "test.db";
my $dsn = "DBI:$driver:dbname=$database";
my $userid = "";
my $password = "";
my $dbh = DBI->connect($dsn, $userid, $password, { RaiseError => 1
                      or die $DBI::errstr;

print "Opened database successfully\n";
```

现在，让我们来运行上面的程序，在当前目录中创建我们的数据库 **test.db**。您可以根据需要改变路径。保存上面代码到 sqlite.pl 文件中，并按如下所示执行。如果数据库成功创建，那么会显示下面所示的消息：

```
$ chmod +x sqlite.pl
$ ./sqlite.pl
Open database successfully
```

创建表

下面的 Perl 代码段将用于在先前创建的数据库中创建一个表：

```
#!/usr/bin/perl

use DBI;
use strict;

my $driver   = "SQLite";
my $database = "test.db";
my $dsn = "DBI:$driver:dbname=$database";
my $userid = "";
my $password = "";
my $dbh = DBI->connect($dsn, $userid, $password, { RaiseError => 1
                      or die $DBI::errstr;
print "Opened database successfully\n";

my $stmt = qq(CREATE TABLE COMPANY
      (ID INT PRIMARY KEY     NOT NULL,
       NAME           TEXT    NOT NULL,
       AGE            INT     NOT NULL,
       ADDRESS        CHAR(50),
       SALARY         REAL););
my $rv = $dbh->do($stmt);
if($rv < 0){
   print $DBI::errstr;
} else {
   print "Table created successfully\n";
}
$dbh->disconnect();
```

上述程序执行时，它会在 **test.db** 中创建 COMPANY 表，并显示下面所示的消息：

```
Opened database successfully
Table created successfully
```

注意：如果您在任何操作中遇到了下面的错误： in case you see following error in any of the operation:

```
DBD::SQLite::st execute failed: not an error(21) at dbdimp.c line 3
```

在这种情况下，您已经在 DBD-SQLite 安装中打开了可用的 dbdimp.c 文件，找到 **sqlite3_prepare()** 函数，并把它的第三个参数 0 改为 -1。最后使用 **make** 和 **make install** 安装 DBD::SQLite，即可解决问题。 in this case you will have open dbdimp.c file available in DBD-SQLite installation and find out **sqlite3_prepare()** function and change its third argument to -1 instead of 0. Finally install DBD::SQLite using **make** and do **make install** to resolve the problem.

INSERT 操作

下面的 Perl 程序显示了如何在上面创建的 COMPANY 表中创建记录：

```
#!/usr/bin/perl

use DBI;
use strict;

my $driver   = "SQLite";
my $database = "test.db";
my $dsn = "DBI:$driver:dbname=$database";
my $userid = "";
my $password = "";
my $dbh = DBI->connect($dsn, $userid, $password, { RaiseError => 1
                      or die $DBI::errstr;
print "Opened database successfully\n";

my $stmt = qq(INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
      VALUES (1, 'Paul', 32, 'California', 20000.00 ));
my $rv = $dbh->do($stmt) or die $DBI::errstr;

$stmt = qq(INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
      VALUES (2, 'Allen', 25, 'Texas', 15000.00 ));
$rv = $dbh->do($stmt) or die $DBI::errstr;

$stmt = qq(INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
      VALUES (3, 'Teddy', 23, 'Norway', 20000.00 ));
$rv = $dbh->do($stmt) or die $DBI::errstr;

$stmt = qq(INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)
      VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 ););
$rv = $dbh->do($stmt) or die $DBI::errstr;

print "Records created successfully\n";
$dbh->disconnect();
```

上述程序执行时，它会在 COMPANY 表中创建给定记录，并会显示以下两行：

```
Opened database successfully
Records created successfully
```

SELECT 操作

下面的 Perl 程序显示了如何从前面创建的 COMPANY 表中获取并显示记录：

```
#!/usr/bin/perl

use DBI;
use strict;

my $driver   = "SQLite";
my $database = "test.db";
my $dsn = "DBI:$driver:dbname=$database";
my $userid = "";
my $password = "";
my $dbh = DBI->connect($dsn, $userid, $password, { RaiseError => 1
                      or die $DBI::errstr;
print "Opened database successfully\n";

my $stmt = qq(SELECT id, name, address, salary  from COMPANY;);
my $sth = $dbh->prepare( $stmt );
my $rv = $sth->execute() or die $DBI::errstr;
if($rv < 0){
   print $DBI::errstr;
}
while(my @row = $sth->fetchrow_array()) {
      print "ID = ". $row[0] . "\n";
      print "NAME = ". $row[1] ."\n";
      print "ADDRESS = ". $row[2] ."\n";
      print "SALARY =  ". $row[3] ."\n\n";
}
print "Operation done successfully\n";
$dbh->disconnect();
```

上述程序执行时，它会产生以下结果：

```
Opened database successfully
ID = 1
NAME = Paul
ADDRESS = California
SALARY =  20000

ID = 2
NAME = Allen
ADDRESS = Texas
SALARY =  15000

ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY =  20000

ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY =  65000

Operation done successfully
```

UPDATE 操作

下面的 Perl 代码显示了如何使用 UPDATE 语句来更新任何记录，然后从 COMPANY 表中获取并显示更新的记录：

```
#!/usr/bin/perl

use DBI;
use strict;

my $driver   = "SQLite";
my $database = "test.db";
my $dsn = "DBI:$driver:dbname=$database";
my $userid = "";
my $password = "";
my $dbh = DBI->connect($dsn, $userid, $password, { RaiseError => 1
                      or die $DBI::errstr;
print "Opened database successfully\n";

my $stmt = qq(UPDATE COMPANY set SALARY = 25000.00 where ID=1;);
my $rv = $dbh->do($stmt) or die $DBI::errstr;
if( $rv < 0 ){
   print $DBI::errstr;
}else{
   print "Total number of rows updated : $rv\n";
}
$stmt = qq(SELECT id, name, address, salary  from COMPANY;);
my $sth = $dbh->prepare( $stmt );
$rv = $sth->execute() or die $DBI::errstr;
if($rv < 0){
   print $DBI::errstr;
}
while(my @row = $sth->fetchrow_array()) {
      print "ID = ". $row[0] . "\n";
      print "NAME = ". $row[1] ."\n";
      print "ADDRESS = ". $row[2] ."\n";
      print "SALARY =  ". $row[3] ."\n\n";
}
print "Operation done successfully\n";
$dbh->disconnect();
```

上述程序执行时，它会产生以下结果：

```
Opened database successfully
Total number of rows updated : 1
ID = 1
NAME = Paul
ADDRESS = California
SALARY =  25000

ID = 2
NAME = Allen
ADDRESS = Texas
SALARY =  15000

ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY =  20000

ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY =  65000

Operation done successfully
```

DELETE 操作

下面的 Perl 代码显示了如何使用 DELETE 语句删除任何记录，然后从 COMPANY 表中获取并显示剩余的记录：

```
#!/usr/bin/perl

use DBI;
use strict;

my $driver   = "SQLite";
my $database = "test.db";
my $dsn = "DBI:$driver:dbname=$database";
my $userid = "";
my $password = "";
my $dbh = DBI->connect($dsn, $userid, $password, { RaiseError => 1
                      or die $DBI::errstr;
print "Opened database successfully\n";

my $stmt = qq(DELETE from COMPANY where ID=2;);
my $rv = $dbh->do($stmt) or die $DBI::errstr;
if( $rv < 0 ){
   print $DBI::errstr;
}else{
   print "Total number of rows deleted : $rv\n";
}
$stmt = qq(SELECT id, name, address, salary  from COMPANY;);
my $sth = $dbh->prepare( $stmt );
$rv = $sth->execute() or die $DBI::errstr;
if($rv < 0){
   print $DBI::errstr;
}
while(my @row = $sth->fetchrow_array()) {
      print "ID = ". $row[0] . "\n";
      print "NAME = ". $row[1] ."\n";
      print "ADDRESS = ". $row[2] ."\n";
      print "SALARY =  ". $row[3] ."\n\n";
}
print "Operation done successfully\n";
$dbh->disconnect();
```

上述程序执行时，它会产生以下结果：

```
Opened database successfully
Total number of rows deleted : 1
ID = 1
NAME = Paul
ADDRESS = California
SALARY =  25000

ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY =  20000

ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY =  65000

Operation done successfully
```

SQLite - Python

安装

SQLite3 可使用 sqlite3 模块与 Python 进行集成。sqlite3 模块是由 Gerhard Haring 编写的。它提供了一个与 PEP 249 描述的 DB-API 2.0 规范兼容的 SQL 接口。您不需要单独安装该模块，因为 Python 2.5.x 以上版本默认自带了该模块。

为了使用 sqlite3 模块，您首先必须创建一个表示数据库的连接对象，然后您可以有选择地创建光标对象，这将帮助您执行所有的 SQL 语句。

Python sqlite3 模块 API

以下是重要的 sqlite3 模块程序，可以满足您在 Python 程序中使用 SQLite 数据库的需求。如果您需要了解更多细节，请查看 Python sqlite3 模块的官方文档。

API	描述
sqlite3.connect(database [,timeout ,other optional arguments])	该 API 打开一个到 SQLite 数据库文件 database 的链接。您可以使用 ":memory:" 来在 RAM 中打开一个到 database 的数据库连接，而不是在磁盘上打开。如果数据库成功打开，则返回一个连接对象。 当一个数据库被多个连接访问，且其中一个修改了数据库，此时 SQLite 数据库被锁定，直到事务提交。timeout 参数表示连接等待锁定的持续时间，直到发生异常断开连接。timeout 参数默认是 5.0（5 秒）。 如果给定的数据库名称 filename 不存在，则该调用将创建一个数据库。如果您不想在当前目录中创建数据库，那么您可以指定带有路径的文件名，这样您就能在任意地方创建数据库。
connection.cursor([cursorClass])	该例程创建一个 **cursor**，将在 Python 数据库编程中用到。该方法接受一个单一的可选的参数 cursorClass。如果提供了

	该参数，则它必须是一个扩展自 sqlite3.Cursor 的自定义的 cursor 类。
cursor.execute(sql [, optional parameters])	该例程执行一个 SQL 语句。该 SQL 语句可以被参数化（即使用占位符代替 SQL 文本）。sqlite3 模块支持两种类型的占位符：问号和命名占位符（命名样式）。 例如：cursor.execute("insert into people values (?, ?)", (who, age))
connection.execute(sql [, optional parameters])	该例程是上面执行的由光标（cursor）对象提供的方法的快捷方式，它通过调用光标（cursor）方法创建了一个中间的光标对象，然后通过给定的参数调用光标的 execute 方法。
cursor.executemany(sql, seq_of_parameters)	该例程对 seq_of_parameters 中的所有参数或映射执行一个 SQL 命令。
connection.executemany(sql[, parameters])	该例程是一个由调用光标（cursor）方法创建的中间的光标对象的快捷方式，然后通过给定的参数调用光标的 executemany 方法。
cursor.executescript(sql_script)	该例程一旦接收到脚本，会执行多个 SQL 语句。它首先执行 COMMIT 语句，然后执行作为参数传入的 SQL 脚本。所有的 SQL 语句应该用分号（;）分隔。
connection.executescript(sql_script)	该例程是一个由调用光标（cursor）方法创建的中间的光标对象的快捷方式，然后通过给定的参数调用光标的 executescript 方法。
connection.total_changes()	该例程返回自数据库连接打开以来被修改、插入或删除的数据库总行数。
	该方法提交当前的食物。如果您未调用该方法，那么自您上

connection.commit()	一次调用 commit() 以来所做的任何动作对其他数据库连接来说是不可见的。
connection.rollback()	该方法回滚自上一次调用 commit() 以来对数据库所做的更改。
connection.close()	该方法关闭数据库连接。请注意，这不会自动调用 commit()。如果您之前未调用 commit() 方法，就直接关闭数据库连接，您所做的所有更改将全部丢失！
cursor.fetchone()	该方法获取查询结果集中的下一行，返回一个单一的序列，当没有更多可用的数据时，则返回 None。
cursor.fetchmany([size=cursor.arraysize])	该方法获取查询结果集中的下一行组，返回一个列表。当没有更多的可用的行时，则返回一个空的列表。该方法尝试获取由 size 参数指定的尽可能多的行。
cursor.fetchall()	该例程获取查询结果集中所有（剩余）的行，返回一个列表。当没有可用的行时，则返回一个空的列表。

连接数据库

下面的 Python 代码显示了如何连接到一个现有的数据库。如果数据库不存在，那么它就会被创建，最后将返回一个数据库对象。

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')

print "Opened database successfully";
```

在这里，您也可以把数据库名称复制为特定的名称 **:memory:**，这样就会在 RAM 中创建一个数据库。现在，让我们来运行上面的程序，在当前目录中创建我们的数据库 **test.db**。您可以根据需要改变路径。保存上面代码到 sqlite.py 文件中，并按

如下所示执行。如果数据库成功创建，那么会显示下面所示的消息：

```
$chmod +x sqlite.py
$./sqlite.py
Open database successfully
```

创建表

下面的 Python 代码段将用于在先前创建的数据库中创建一个表：

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

conn.execute('''CREATE TABLE COMPANY
       (ID INT PRIMARY KEY     NOT NULL,
       NAME           TEXT    NOT NULL,
       AGE            INT     NOT NULL,
       ADDRESS        CHAR(50),
       SALARY         REAL);''')
print "Table created successfully";

conn.close()
```

上述程序执行时，它会在 **test.db** 中创建 COMPANY 表，并显示下面所示的消息：

```
Opened database successfully
Table created successfully
```

INSERT 操作

下面的 Python 程序显示了如何在上面创建的 COMPANY 表中创建记录：

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (1, 'Paul', 32, 'California', 20000.00 )");

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (2, 'Allen', 25, 'Texas', 15000.00 )");

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (3, 'Teddy', 23, 'Norway', 20000.00 )");

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 )");

conn.commit()
print "Records created successfully";
conn.close()
```

上述程序执行时，它会在 COMPANY 表中创建给定记录，并会显示以下两行：

```
Opened database successfully
Records created successfully
```

SELECT 操作

下面的 Python 程序显示了如何从前面创建的 COMPANY 表中获取并显示记录：

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

cursor = conn.execute("SELECT id, name, address, salary  from COMPA
for row in cursor:
   print "ID = ", row[0]
   print "NAME = ", row[1]
   print "ADDRESS = ", row[2]
   print "SALARY = ", row[3], "\n"

print "Operation done successfully";
conn.close()
```

上述程序执行时，它会产生以下结果：

```
Opened database successfully
ID =  1
NAME =  Paul
ADDRESS =  California
SALARY =  20000.0

ID =  2
NAME =  Allen
ADDRESS =  Texas
SALARY =  15000.0

ID =  3
NAME =  Teddy
ADDRESS =  Norway
SALARY =  20000.0

ID =  4
NAME =  Mark
ADDRESS =  Rich-Mond
SALARY =  65000.0

Operation done successfully
```

UPDATE 操作

下面的 Python 代码显示了如何使用 UPDATE 语句来更新任何记录，然后从 COMPANY 表中获取并显示更新的记录：

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

conn.execute("UPDATE COMPANY set SALARY = 25000.00 where ID=1")
conn.commit
print "Total number of rows updated :", conn.total_changes

cursor = conn.execute("SELECT id, name, address, salary  from COMPA
for row in cursor:
   print "ID = ", row[0]
   print "NAME = ", row[1]
   print "ADDRESS = ", row[2]
   print "SALARY = ", row[3], "\n"

print "Operation done successfully";
conn.close()
```

上述程序执行时，它会产生以下结果：

```
Opened database successfully
Total number of rows updated : 1
ID =  1
NAME =  Paul
ADDRESS =  California
SALARY =  25000.0

ID =  2
NAME =  Allen
ADDRESS =  Texas
SALARY =  15000.0

ID =  3
NAME =  Teddy
ADDRESS =  Norway
SALARY =  20000.0

ID =  4
NAME =  Mark
ADDRESS =  Rich-Mond
SALARY =  65000.0

Operation done successfully
```

DELETE 操作

下面的 Python 代码显示了如何使用 DELETE 语句删除任何记录，然后从 COMPANY 表中获取并显示剩余的记录：

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

conn.execute("DELETE from COMPANY where ID=2;")
conn.commit
print "Total number of rows deleted :", conn.total_changes

cursor = conn.execute("SELECT id, name, address, salary  from COMPA
for row in cursor:
   print "ID = ", row[0]
   print "NAME = ", row[1]
   print "ADDRESS = ", row[2]
   print "SALARY = ", row[3], "\n"

print "Operation done successfully";
conn.close()
```

上述程序执行时，它会产生以下结果：

```
Opened database successfully
Total number of rows deleted : 1
ID =  1
NAME =  Paul
ADDRESS =  California
SALARY =  20000.0

ID =  3
NAME =  Teddy
ADDRESS =  Norway
SALARY =  20000.0

ID =  4
NAME =  Mark
ADDRESS =  Rich-Mond
SALARY =  65000.0

Operation done successfully
```

W3School MangoDB教程

来源：MongoDB教程

整理：飞龙

NoSQL 简介

NoSQL(NoSQL = Not Only SQL)，意即"不仅仅是SQL"。

在现代的计算系统上每天网络上都会产生庞大的数据量。

这些数据有很大一部分是由关系数据库管理系统（RDMBSs）来处理。 1970年 E.F.Codd's提出的关系模型的论文 "A relational model of data for large shared data banks"，这使得数据建模和应用程序编程更加简单。

通过应用实践证明，关系模型是非常适合于客户服务器编程，远远超出预期的利益，今天它是结构化数据存储在网络和商务应用的主导技术。

NoSQL 是一项全新的数据库革命性运动，早期就有人提出，发展至2009年趋势越发高涨。NoSQL的拥护者们提倡运用非关系型的数据存储，相对于铺天盖地的关系型数据库运用，这一概念无疑是一种全新的思维的注入。

关系型数据库遵循ACID规则

事务在英文中是transaction，和现实世界中的交易很类似，它有如下四个特性：

1、A (Atomicity) 原子性
原子性很容易理解，也就是说事务里的所有操作要么全部做完，要么都不做，事务成功的条件是事务里的所有操作都成功，只要有一个操作失败，整个事务就失败，需要回滚。

比如银行转账，从A账户转100元至B账户，分为两个步骤：1）从A账户取100元；2）存入100元至B账户。这两步要么一起完成，要么一起不完成，如果只完成第一步，第二步失败，钱会莫名其妙少了100元。

2、C (Consistency) 一致性
一致性也比较容易理解，也就是说数据库要一直处于一致的状态，事务的运行不会改变数据库原本的一致性约束。

例如现有完整性约束a+b=10，如果一个事务改变了a，那么必须得改变b，使得事务结束后依然满足a+b=10，否则事务失败。

3、I (Isolation) 独立性
所谓的独立性是指并发的事务之间不会互相影响，如果一个事务要访问的数据正在被另外一个事务修改，只要另外一个事务未提交，它所访问的数据就不受未提交事务的影响。
比如现有有个交易是从A账户转100元至B账户，在这个交易还未完成的情况下，如果此时B查询自己的账户，是看不到新增加的100元的。

4、D (Durability) 持久性
持久性是指一旦事务提交后，它所做的修改将会永久的保存在数据库上，即使出现宕机也不会丢失。

分布式系统

分布式系统（distributed system）由多台计算机和通信的软件组件通过计算机网络连接（本地网络或广域网）组成。

分布式系统是建立在网络之上的软件系统。正是因为软件的特性，所以分布式系统具有高度的内聚性和透明性。

因此，网络和分布式系统之间的区别更多的在于高层软件（特别是操作系统），而不是硬件。

分布式系统可以应用在在不同的平台上如：Pc、工作站、局域网和广域网上等。

分布式计算的优点

可靠性（容错）：
分布式计算系统中的一个重要的优点是可靠性。一台服务器的系统崩溃并不影响到其余的服务器。

可扩展性：
在分布式计算系统可以根据需要增加更多的机器。

资源共享：
共享数据是必不可少的应用，如银行，预订系统。

灵活性：
由于该系统是非常灵活的，它很容易安装，实施和调试新的服务。

更快的速度：
分布式计算系统可以有多台计算机的计算能力，使得它比其他系统有更快的处理速度。

开放系统：
由于它是开放的系统，本地或者远程都可以访问到该服务。

更高的性能：
相较于集中式计算机网络集群可以提供更高的性能（及更好的性价比）。

分布式计算的缺点

故障排除：：
故障排除和诊断问题。

软件：
更少的软件支持是分布式计算系统的主要缺点。

网络：
网络基础设施的问题，包括：传输问题，高负载，信息丢失等。

安全性：
开发系统的特性让分布式计算系统存在着数据的安全性和共享的风险等问题。

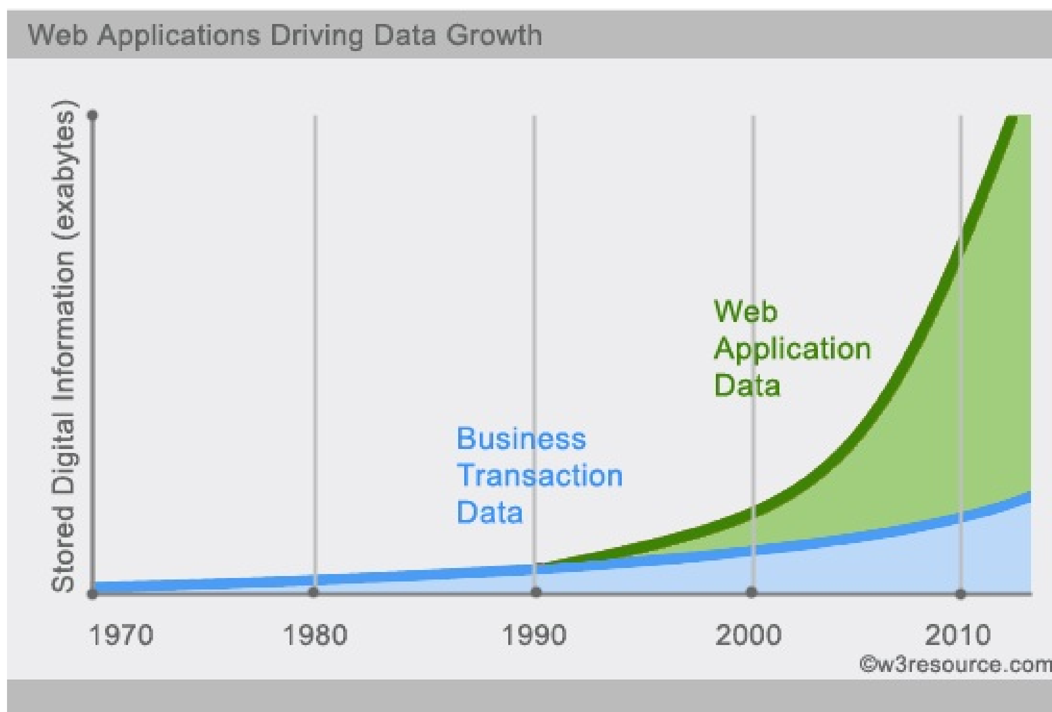
什么是NoSQL?

NoSQL，指的是非关系型的数据库。NoSQL有时也称作Not Only SQL的缩写，是对不同于传统的关系型数据库的数据库管理系统的统称。

NoSQL用于超大规模数据的存储。（例如谷歌或Facebook每天为他们的用户收集万亿比特的数据）。这些类型的数据存储不需要固定的模式，无需多余操作就可以横向扩展。

为什么使用NoSQL ?

今天我们可以通过第三方平台（如：Google,Facebook等）可以很容易的访问和抓取数据。用户的个人信息，社交网络，地理位置，用户生成的数据和用户操作日志已经成倍的增加。我们如果要对这些用户数据进行挖掘，那SQL数据库已经不适合这些应用了, NoSQL数据库的发展也却能很好的处理这些大的数据。



实例

社会化关系网:

```
Each record: UserID1, UserID2
Separate records: UserID, first_name,last_name, age, gender,...
Task: Find all friends of friends of friends of ... friends of a gi
```

Wikipedia 页面：

```
Large collection of documents
Combination of structured and unstructured data
Task: Retrieve all pages regarding athletics of Summer Olympic befo
```

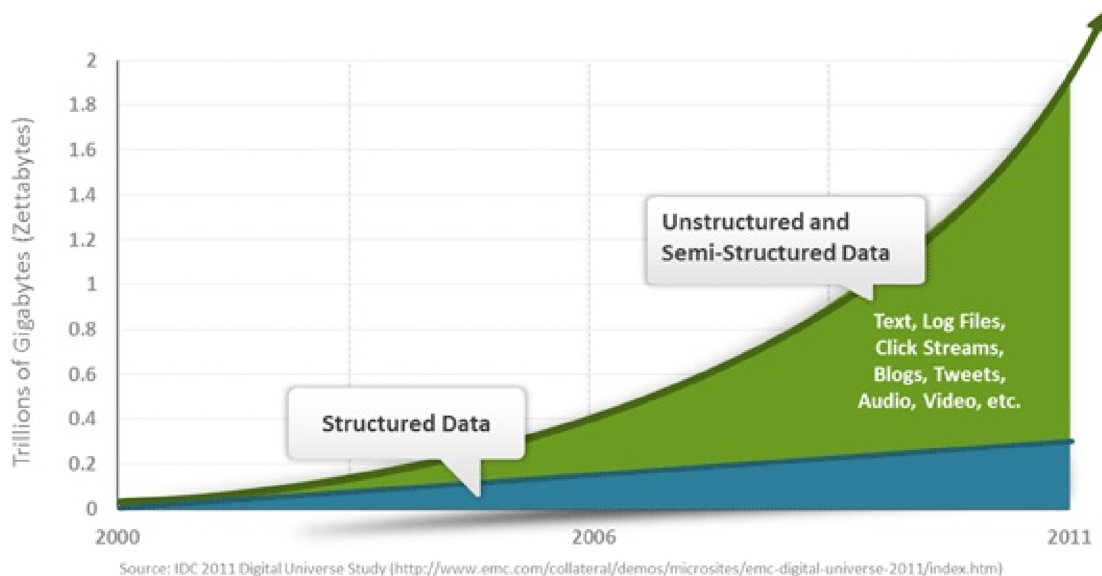
RDBMS vs NoSQL

RDBMS

- 高度组织化结构化数据
- 结构化查询语言（SQL） (SQL)
- 数据和关系都存储在单独的表中。
- 数据操纵语言，数据定义语言
- 严格的一致性
- 基础事务

NoSQL

- 代表着不仅仅是SQL
- 没有声明性查询语言
- 没有预定义的模式
  -键 - 值对存储，列存储，文档存储，图形数据库
- 最终一致性，而非ACID属性
- 非结构化和不可预知的数据
- CAP定理
- 高性能，高可用性和可伸缩性



NoSQL 简史

NoSQL一词最早出现于1998年，是Carlo Strozzi开发的一个轻量、开源、不提供SQL功能的关系数据库。

2009年，Last.fm的Johan Oskarsson发起了一次关于分布式开源数据库的讨论[2]，来自Rackspace的Eric Evans再次提出了NoSQL的概念，这时的NoSQL主要指非关系型、分布式、不提供ACID的数据库设计模式。

2009年在亚特兰大举行的"no:sql(east)"讨论会是一个里程碑，其口号是"select fun, profit from real_world where relational=false;"。因此，对NoSQL最普遍的解释是"非关联型的"，强调Key-Value Stores和文档数据库的优点，而不是单纯的反对RDBMS。

CAP定理（CAP theorem）

在计算机科学中, CAP定理（CAP theorem）, 又被称作 布鲁尔定理（Brewer's theorem）, 它指出对于一个分布式计算系统来说，不可能同时满足以下三点:

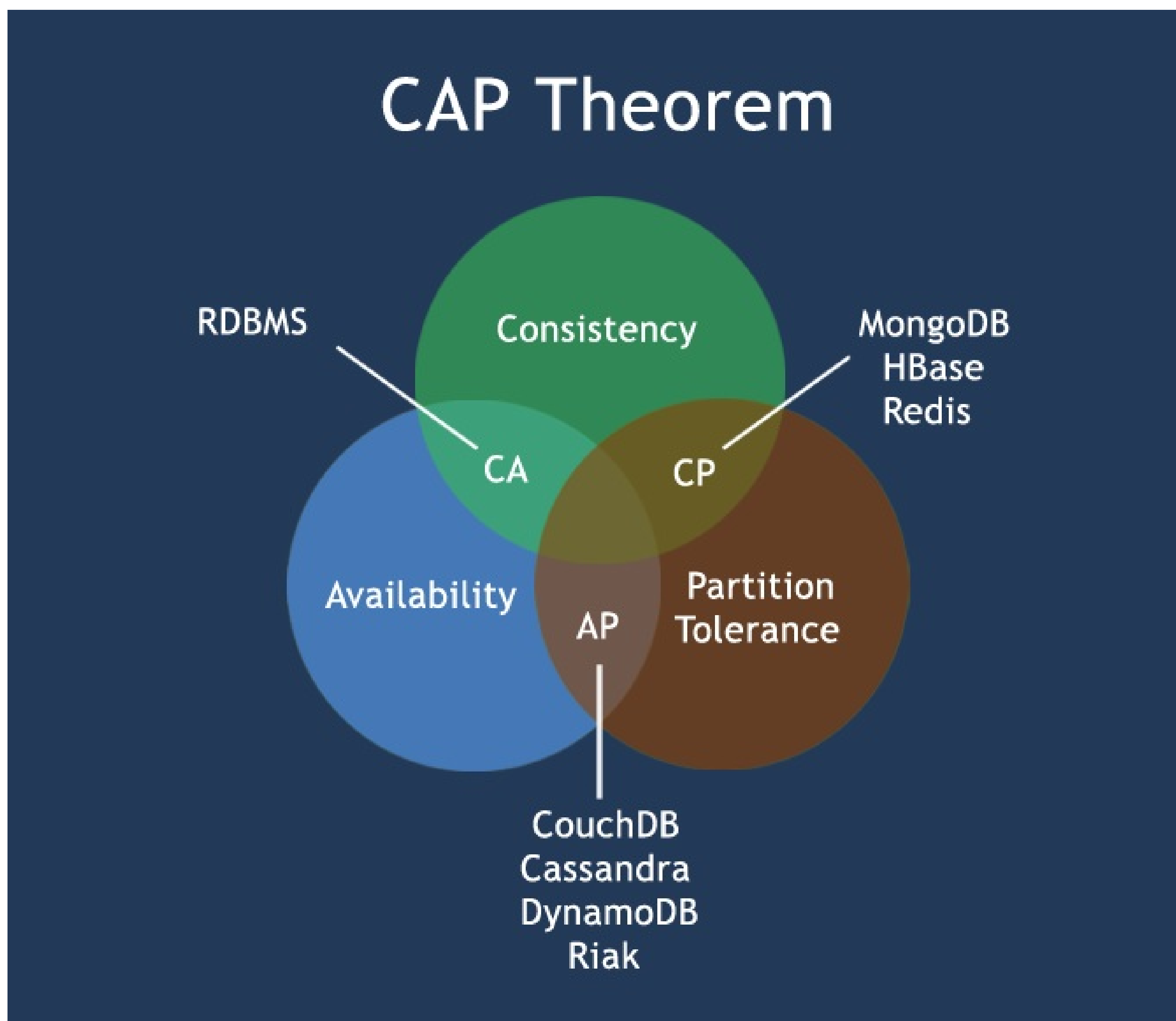
- 一致性(**Consistency**) (所有节点在同一时间具有相同的数据)
- 可用性(**Availability**) (保证每个请求不管成功或者失败都有响应)
- 分隔容忍(**Partition tolerance**) (系统中任意信息的丢失或失败不会影响系统的继续运作)

CAP理论的核心是：一个分布式系统不可能同时很好的满足一致性，可用性和分区容错性这三个需求，最多只能同时较好的满足两个。

因此，根据 CAP 原理将 NoSQL 数据库分成了满足 CA 原则、满足 CP 原则和满足 AP 原则三 大类：

- CA - 单点集群，满足一致性，可用性的系统，通常在可扩展性上不太强大。

- CP - 满足一致性，分区容忍必的系统，通常性能不是特别高。
- AP - 满足可用性，分区容忍性的系统，通常可能对一致性要求低一些。



NoSQL的优点/缺点

优点:

-
    - 高可扩展性
-
    - 分布式计算
-
    - 低成本
-
    - 架构的灵活性，半结构化数据
-
    - 没有复杂的关系

缺点:

-
  - 没有标准化
-
  - 有限的查询功能（到目前为止）
-
  - 最终一致是不直观的程序

BASE

BASE：Basically Available, Soft-state, Eventually Consistent。 由 Eric Brewer 定义。

CAP理论的核心是：一个分布式系统不可能同时很好的满足一致性，可用性和分区容错性这三个需求，最多只能同时较好的满足两个。

BASE是NoSQL数据库通常对可用性及一致性的弱要求原则:

- Basically Availble --基本可用
- Soft-state --软状态/柔性事务。 "Soft state" 可以理解为"无连接"的, 而 "Hard state" 是"面向连接"的
- Eventual Consistency --最终一致性 最终一致性， 也是是 ACID 的最终目的。

ACID vs BASE

ACID	BASE
原子性(**A**tomicity)	基本可用(**B**asically **A**vailable)
一致性(**C**onsistency)	软状态/柔性事务(**S**oft state)
隔离性(**I**solation)	最终一致性 (**E**ventual consistency)
持久性 (**D**urable)	

NoSQL 数据库分类

类型	部分代表	特点
列存储	Hbase Cassandra Hypertable	顾名思义，是按列存储数据的。最大的特点是方便存储结构化和半结构化数据，方便做数据压缩，对针对某一列或者某几列的查询有非常大的IO优势。
文档存储	MongoDB CouchDB	文档存储一般用类似json的格式存储，存储的内容是文档型的。这样也就有有机会对某些字段建立索引，实现关系数据库的某些功能。
key-value存储	Tokyo Cabinet / Tyrant Berkeley DB MemcacheDB Redis	可以通过key快速查询到其value。一般来说，存储不管value的格式，照单全收。（Redis包含了其他功能）
图存储	Neo4J FlockDB	图形关系的最佳存储。使用传统关系数据库来解决的话性能低下，而且设计使用不方便。
对象存储	db4o Versant	通过类似面向对象语言的语法操作数据库，通过对象的方式存取数据。
xml数据库	Berkeley DB XML BaseX	高效的存储XML数据，并支持XML的内部查询语法，比如XQuery,Xpath。

谁在使用

现在已经有很多公司使用了NoSQ：

- Google
- Facebook
- Mozilla
- Adobe
- Foursquare
- LinkedIn
- Digg
- McGraw-Hill Education
- Vermont Public Radio

什么是MongoDB ?

MongoDB 是由C++语言编写的开源数据库系统。

在高负载的情况下，添加更多的节点，可以保证服务器性能。

MongoDB 旨在为WEB应用提供可扩展的高性能数据存储解决方案。



MongoDB 将数据存储为一个文档。MongoDB是一个基于分布式文件存储的数据库。

```
FirstName="Arun", Address="St. Xavier's Road", Spouse=[{Name:"Kiran
FirstName="Sameer",Address="8 Gandhi Road".
```

注意：以上数据有两个不同的文档（以"."分隔）。以这种方式存储数据即为文件存储的数据库。 MongoDB是一个面向文档的数据库。

主要特点

- MongoDB的提供了一个面向文档存储，操作起来比较简单和容易。
- 你可以在MongoDB记录中设置任何属性的索引 (如：FirstName="Sameer",Address="8 Gandhi Road")来实现更快的排序。
- 你可以通过本地或者网络创建数据镜像，这使得MongoDB有更强的扩展性。
- 如果负载的增加（需要更多的存储空间和更强的处理能力），它可以分布在计算机网络中的其他节点上这就是所谓的分片。
- Mongo支持丰富的查询表达式。查询指令使用JSON形式的标记，可轻易查询文档中内嵌的对象及数组。
- MongoDb 使用update()命令可以实现替换完成的文档（数据）或者一些指定的数据字段 。
- Mongodb中的Map/reduce主要是用来对数据进行批量处理和聚合操作。
- Map和Reduce。Map函数调用emit(key,value)遍历集合中所有的记录，将key与value传给Reduce函数进行处理。
- Map函数和Reduce函数是使用Javascript编写的，并可以通过db.runCommand或mapreduce命令来执行MapReduce操作。
- GridFS是MongoDB中的一个内置功能，可以用于存放大量小文件。
- MongoDB允许在服务端执行脚本，可以用Javascript编写某个函数，直接在服务端执行，也可以把函数的定义存储在服务端，下次直接调用即可。
- MongoDB支持各种编程语言:RUBY，PYTHON，JAVA，C++，PHP，C#等多

种语言。
- MongoDB安装简单。

历史 2007年10月，MongoDB由10gen团队所发展。2009年2月首度推出。

- 2012年05月23日，MongoDB2.1 开发分支发布了! 该版本采用全新架构，包含诸多增强。
- 2012年06月06日，MongoDB 2.0.6 发布，分布式文档数据库。
- 2013年04月23日，MongoDB 2.4.3 发布，此版本包括了一些性能优化，功能增强以及bug修复。
- 2013年08月20日，MongoDB 2.4.6 发布，是目前最新的稳定版。

MongoDB 下载

你可以在mongodb官网下载该安装包，地址为：http://www.mongodb.org/downloads。MonggoDB支持以下平台:

- OS X 32-bit
- OS X 64-bit
- Linux 32-bit
- Linux 64-bit
- Windows 32-bit
- Windows 64-bit
- Solaris i86pc
- Solaris 64

MongoDB 工具

有几种可用于MongoDB的管理工具。

监控

MongoDB提供了网络和系统监控工具Munin，它作为一个插件应用于MongoDB中。

Gangila是MongoDB高性能的系统监视的工具，它作为一个插件应用于MongoDB中。

基于图形界面的开源工具 Cacti, 用于查看CPU负载, 网络带宽利用率,它也提供了一个应用于监控 MongoDB 的插件。

GUI

- Fang of Mongo – 网页式,由Django和jQuery所构成。
- Futon4Mongo – 一个CouchDB Futon web的mongodb山寨版。
- Mongo3 – Ruby写成。
- MongoHub – 适用于OSX的应用程序。
- Opricot – 一个基于浏览器的MongoDB控制台, 由PHP撰写而成。

- Database Master — Windows的mongodb管理工具
- RockMongo — 最好的PHP语言的MongoDB管理工具，轻量级, 支持多国语言.

MongoDB 应用案例

下面列举一些公司MongoDB的实际应用：

- Craiglist上使用MongoDB的存档数十亿条记录。
- FourSquare，基于位置的社交网站，在Amazon EC2的服务器上使用MongoDB分享数据。
- Shutterfly，以互联网为基础的社会和个人出版服务，使用MongoDB的各种持久性数据存储的要求。
- bit.ly, 一个基于Web的网址缩短服务，使用MongoDB的存储自己的数据。
- spike.com，一个MTV网络的联营公司， spike.com使用MongoDB的。
- Intuit公司，一个为小企业和个人的软件和服务提供商，为小型企业使用MongoDB的跟踪用户的数据。
- sourceforge.net，资源网站查找，创建和发布开源软件免费，使用MongoDB的后端存储。
- etsy.com ，一个购买和出售手工制作物品网站，使用MongoDB。
- 纽约时报，领先的在线新闻门户网站之一，使用MongoDB。
- CERN，著名的粒子物理研究所，欧洲核子研究中心大型强子对撞机的数据使用MongoDB。

window平台安装 MongoDB

MongoDB 下载

MongoDB提供了可用于32位和64位系统的预编译二进制包，你可以从MongoDB官网下载安装，MongoDB预编译二进制包下载地址：

http://www.mongodb.org/downloads

	OS X 32-bit note	OS X 64-bit	Linux 32-bit note	Linux 64-bit	Windows 32-bit note	Windows 64-bit	Solaris i86pc note	Solaris 64	Source
Production Release (Recommended)									
1.8.2 6/17/2011 Changelog Release Notes	download	download	download *legacy-static	download *legacy-static	download	download	download	download	tgz zip

解压

下载zip包后，解压安装包，并安装它。

创建数据目录

MongoDB将数据目录存储在 db 目录下。但是这个数据目录不会主动创建，我们在安装完成后需要创建它。请注意，数据目录应该抽奖在根目录下（(如： C:\ 或者 D:\ 等)。

在本教程中，我们已经在D：盘中解压了mongodb文件，现在让我们创建一个data的目录然后在data目录里创建db目录。

```
D:\>mkdir data
D:\>cd data
D:\data>mkdir db
D:\data>cd db
D:\data\db>
```

你也可以通过window的资源管理器中创建这些目录，而不一定通过命令行。

命令行下运行 MongoDB 服务器

为了从命令提示符下运行MongoDB服务器，你必须从MongoDB目录的bin目录中执行mongod.exe文件。

```
D:\mongodb>cd bin

D:\mongodb\bin>mongod
mongod --help for help and startup options
Thu Jul 07 12:59:48 [initandlisten] MongoDB starting : pid=2448 port=27017 dbpat
h=/data/db/ 32-bit

** NOTE: when using MongoDB 32 bit, you are limited to about 2 gigabytes of data

**       see http://blog.mongodb.org/post/137788967/32-bit-limitations
**       with --dur, the limit is lower

Thu Jul 07 12:59:48 [initandlisten] db version v1.8.1, pdfile version 4.5
Thu Jul 07 12:59:48 [initandlisten] git version: a429cd4f535b2499cc4130b06ff7c26
f41c00f04
Thu Jul 07 12:59:48 [initandlisten] build sys info: windows (5, 1, 2600, 2, 'Ser
vice Pack 3') BOOST_LIB_VERSION=1_35
Thu Jul 07 12:59:49 [initandlisten] waiting for connections on port 27017
Thu Jul 07 12:59:49 [websvr] web admin interface listening on port 28017
```

将MongoDB服务器作为Windows服务运行

请注意，你必须有管理权限才能运行下面的命令。执行以下命令将MongoDB服务器作为Windows服务运行：

mongod --bind_ip yourIPadress --logpath "C:\data\dbConf\mongodb.log" --logappend --dbpath "C:\data\db" --port yourPortNumber --serviceName "YourServiceName" --serviceDisplayName "YourServiceName" --install

下表为**mongodb**启动的参数说明：

参数	描述
--bind_ip	绑定服务IP，若绑定127.0.0.1，则只能本机访问，不指定默认本地所有IP
--logpath	定MongoDB日志文件，注意是指定文件不是目录
--logappend	使用追加的方式写日志
--dbpath	指定数据库路径
--port	指定服务端口号，默认端口27017
--serviceName	指定服务名称
--serviceDisplayNam	指定服务名称，有多个mongodb服务时执行。
--install	指定作为一个Windows服务安装。

MongoDB后台管理 Shell

如果你需要进入MongoDB后台管理，你需要先打开mongodb装目录的下的bin目录，然后执行mongo.exe文件，MongoDB Shell是MongoDB自带的交互式Javascript shell,用来对MongoDB进行操作和管理的交互式环境。

当你进入mongoDB后台后，它默认会链接到 test 文档（数据库）：

```
D:\>cd mongodb

D:\mongodb>cd bin

D:\mongodb\bin>mongo
MongoDB shell version: 1.8.1
connecting to: test
>
```

由于它是一个JavaScript shell，您可以运行一些简单的算术运算:

```
D:\>cd nongodb

D:\mongodb>cd bin

D:\mongodb\bin>mongo
MongoDB shell version: 1.8.1
connecting to: test
> 2+2
4
>
```

db 命令先了当前操作的文档（数据库）：

```
D:\>cd mongodb

D:\mongodb>cd bin

D:\mongodb\bin>mongo
MongoDB shell version: 1.8.1
connecting to: test
> 2+2
4
> db
test
>
```

插入一些简单的记录并查找它：

```
D:\mongodb\bin>mongo
MongoDB shell version: 1.8.1
connecting to: test
> db.w3r.insert({x:10})
> db.w3r.find()
{ "_id" : ObjectId("4e156fdee1aa39b8e1e77fad"), "x" : 10 }
>
```

第一个命令将10插入到w3r集合的x字段中

Linux平台安装MongoDB

下载

MongoDB提供了linux平台上32位和64位的安装包，你可以在官网下载安装包。

下载地址：http://www.mongodb.org/downloads

	OS X 32-bit note	OS X 64-bit	Linux 32-bit note	Linux 64-bit	Windows 32-bit note	Windows 64-bit	Solaris i86pc note	Solaris 64	Source
Production Release (Recommended)									
1.8.2 6/17/2011 Changelog Release Notes	download	download	download *legacy-static	download *legacy-static	download	download	download	download	tgz zip

安装

下载完成后，在你安装的目录下解压zip包。

创建数据库目录

MongoDB的数据存储在data目录的db目录下，但是这个目录在安装过程不会自动创建，所以你需要手动创建data目录，并在data目录中创建db目录。

注意：请将data目录创建于根目录下(/)。

```
File  Edit  View  Terminal  Tabs  Help
debian:/# mkdir data
debian:/# cd data
debian:/data# mkdir db
debian:/data# cd db
debian:/data/db#
```

命令行中运行 MongoDB 服务

你可以再命令行中执行mongo安装目录中的bin目录执行mongod命令来启动mongdb服务。

```
debian:/home/ritwik# ./mongodb/bin/mongod
./mongodb/bin/mongod --help for help and startup options
Tue Jul 12 07:59:40 [initandlisten] MongoDB starting : pid=3715 port=27017 dbpath=/data/db/ 32-bit

** NOTE: when using MongoDB 32 bit, you are limited to about 2 gigabytes of data
**       see http://blog.mongodb.org/post/137788967/32-bit-limitations
**       with --dur, the limit is lower

Tue Jul 12 07:59:40 [initandlisten] db version v1.8.2, pdfile version 4.5
Tue Jul 12 07:59:40 [initandlisten] git version: 433bbaa14aaba6860da15bd4de8edf600f56501b
Tue Jul 12 07:59:40 [initandlisten] build sys info: Linux bs-linux32.10gen.cc 2.6.21.7-2.fc8xen #1 SMP Fri Feb
ST_LIB_VERSION=1_37
Tue Jul 12 07:59:40 [initandlisten] waiting for connections on port 27017
Tue Jul 12 07:59:40 [websvr] web admin interface listening on port 28017
```

MongoDB后台管理 Shell

如果你需要进入MongoDB后台管理，你需要先打开mongodb装目录的下的bin目录，然后执行mongo命令文件。

MongoDB Shell是MongoDB自带的交互式Javascript shell,用来对MongoDB进行操作和管理的交互式环境。

当你进入mongoDB后台后，它默认会链接到 test 文档（数据库）：

```
debian:/home/ritwik# ./mongodb/bin/mongod
./mongodb/bin/mongod --help for help and startup options
Tue Jul 12 07:59:40 [initandlisten] MongoDB starting : pid=3715 port=27017 dbpath=/data/db/ 32-bit

** NOTE: when using MongoDB 32 bit, you are limited to about 2 gigabytes of data
**       see http://blog.mongodb.org/post/137788967/32-bit-limitations
**       with --dur, the limit is lower

Tue Jul 12 07:59:40 [initandlisten] db version v1.8.2, pdfile version 4.5
Tue Jul 12 07:59:40 [initandlisten] git version: 433bbaa14aaba6860da15bd4de8edf600f56501b
Tue Jul 12 07:59:40 [initandlisten] build sys info: Linux bs-linux32.10gen.cc 2.6.21.7-2.fc8xen #1 SMP Fri Feb
ST_LIB_VERSION=1_37
Tue Jul 12 07:59:40 [initandlisten] waiting for connections on port 27017
Tue Jul 12 07:59:40 [websvr] web admin interface listening on port 28017
```

由于它是一个JavaScript shell，您可以运行一些简单的算术运算:

```
File  Edit  View  Terminal  Tabs  Help
debian:/home/ritwik# ./mongodb/bin/mongo
MongoDB shell version: 1.8.2
connecting to: test
> 3+5
8
> db
test
>
```

现在让我们插入一些简单的数据，并对插入的数据进行检索：

```
debian:/home/ritwik# ./mongodb/bin/mongo
MongoDB shell version: 1.8.2
connecting to: test
> 3+5
8
> db
test
> db.w3r.insert({z:8})
> db.w3r.find()
{ "_id" : ObjectId("4e1bb52ccb73fc0d2cf72818"), "z" : 8 }
>
```

第一个命令是将数据 8 插入到w3r集合（表）的 z 字段中。

MongoDb web 用户界面

在比MongoDB服务的端口多1000的端口上，你可以访问到MondoDB的web用户界面。

如：如果你的MongoDB运行端口使用默认的27017，你可以在端口号为28017访问web用户界面。



mongod debian

List all commands | Replica set status

Commands: buildinfo cursorInfo features isMaster listDatabases replSetGetStatus serverStatus top

db version v1.8.2, pdfile version 4.5
git hash: 433bbaa14aaba6860da15bd4de8edf600f56501b
sys info: Linux bs-linux32.10gen.cc 2.6.21.7-2.fc8xen #1 SMP Fri Feb 15 12:39:36 EST 2008 i686 BOOST_LIB_VERSION=1_37
uptime: 298 seconds

---

low level requires read lock

time to get readlock: 20ms
databases: 2

replication:
master: 0
slave: 0
initialSyncCompleted: 1

---

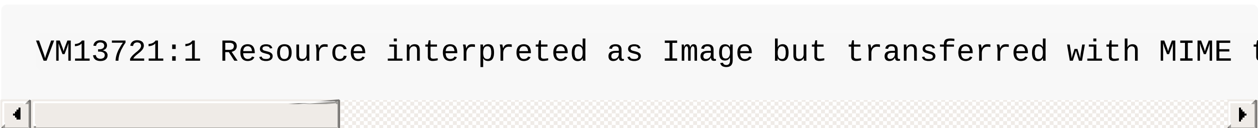
clients

Client	OpId	Active	LockType	Waiting	SecsRunning	Op	Namespace	Query	client	msg	progress
snapshotthread	0		0			0			(NONE)		
clientcursormon	0		R			0			(NONE)		
websvr	0		0			0			(NONE)		
conn	9		0			2004	test.$cmd	{ replSetGetStatus: 1, forShell: 1 }	127.0.0.1:48593		
initandlisten	0		W			2004	local.system.namespaces	{ name: /^local.temp./ }	0.0.0.0:0		

---

dbtop (occurences|percent of elapsed)

NS	total		Reads		Writes		Queries		GetMores		Inserts		Updates		Removes	
TOTAL	3	58.2%	2	0.4%	1	55.8%	2	0.4%	0	0%	1	55.8%	0	0%	0	0%
local.system.namespaces	1	0.0%	1	0.0%	0	0%	1	0.0%	0	0%	0	0%	0	0%	0	0%



```
VM13721:1 Resource interpreted as Image but transferred with MIME t
```

MongoDB 数据库，对象，集合

描述

不管我们学习什么数据库都应该学习其中的基础概念，在mongodb中基本的概念是文档、集合、数据库，下面我们挨个介绍。

数据库

一个mongodb中可以建立多个数据库。

MongoDB的默认数据库为"db"，该数据库存储在data目录中。

在MongoDB中可以创建数据库，如果你想使用MongoDB，创建数据库不是必要的。

"show dbs" 命令可以显示所有数据的列表。

```
D:\mongodb\bin>mongo
MongoDB shell version: 1.8.1
connecting to: test
> show dbs
admin   (empty)
comedy  0.03125GB
local   (empty)
student 0.03125GB
test    0.03125GB
> _
```

执行 "db" 命令可以显示当前数据库对象或者集合。

```
D:\mongodb\bin>mongo
MongoDB shell version: 1.8.1
connecting to: test
> db
test
> _
```

运行"use"命令，可以连接到一个指定的数据库。

```
> db
test
> use student
switched to db student
>
```

以上实例命令中，"student" 是你要检索的数据库。

在下一个章节我们将详细讲解MongoDB中命令的使用。

数据库名称可以是任何字符，但是不能包含空字符串，点号（.），或者" "。

"system" 作为系统保留字符串不能作为数据库名。

数据库名不能包含 "$"。

文档

文档是mongodb中的最核心的概念，是其核心单元，我们可以将文档类比成关系型数据库中的每一行数据。

多个键及其关联的值有序的放置在一起就是文档。在mongodb中使用一种类json的bson存储数据。

bson数据可以理解为在json的基础上添加了一些json中没有的数据类型。

如果我们会json，那么bson我们就已经掌握了一半了，至于新添加的数据类型后面我会介绍。

文档例子如下：

```
{ site : "w3cschool.cc" }
```

通常，"object（对象）" 术语是指一个文件。

文件类似于一个RDBMS的记录。

我们可以对集合（collection）进行插入，更新和删除操作。

下表将帮助您更容易理解Mongo中的一些概念：

RDBMS	MongoDB
Table（表）	Collection（集合）
Column（栏）	Key（键）
Value（值）	Value（值）
Records / Rows（记录/列）	Document / Object（文档/对象）

下表为MongoDB中常用的几种数据类型。

数据类型	描述
string（字符串）	可以是一个空字符串或者字符组合。
integer（整型）	整数。
boolean（布尔型）	逻辑值 True 或者 False。
double	双精度浮点型
null	不是0，也不是空。
array	数组：一系列值
object	对象型，程序中被使用的实体。可以是一个值，变量，函数，或者数据结构。
timestamp	timestamp存储为64为的值，只运行一个mongod时可以确保是唯一的。前32位保存的是UTC时间，单位是秒，后32为是在这一秒内的计数值，从0开始，每新建一个MongoTimestamp对象就加一。
Internationalized Strings	UTF-8 字符串。
Object IDs	在mongodb中的文档需要使用唯一的关键字_id来标识他们。几乎每一个mongodb文档都使用_id字段作为第一个属性（在系统集合和定容量集合（capped collection）中有一些例外）。_id值可以是任何类型，最常见的做法是使用ObjectId类型。

集合

集合就是一组文档的组合。如果将文档类比成数据库中的行，那么集合就可以类比成数据库的表。

在mongodb中的集合是无模式的，也就是说集合中存储的文档的结构可以是不同的，比如下面的两个文档可以同时存入到一个集合中：

```
{"name":"mengxiangyue"} {"Name":"mengxiangyue","sex":"nan"}
```

当第一个文档插入时，集合就会被创建。

合法的集合名

集合名称必须以字母或下划线开头。

集合名可以保护数字

集合名称不能使美元符"$"，"$"是系统保留字符。

集合的名字 最大不能超过128个字符 。

另外，"."号的使用在集合当中是允许的，它们被成为子集合(Subcollection)；比如你有一个blog集合，你可以使用blog.title，blog.content或者blog.author来帮组你更好地组织集合。

如下实例：

```
db.tutorials.php.findOne()
```

capped collections

Capped collections 就是固定大小的collection。

它有很高的性能以及队列过期的特性(过期按照插入的顺序). 有点和 "RRD" 概念类似。

Capped collections是高性能自动的维护对象的插入顺序。它非常适合类似记录日志的功能 和标准的collection不同，你必须要显式的创建一个capped collection，指定一个collection的大小，单位是字节。collection的数据存储空间值提前分配的。

要注意的是指定的存储大小包含了数据库的头信息。> db.createCollection("mycoll", {capped:true, size:100000})

- 在capped collection中，你能添加新的对象。
- 能进行更新，然而，对象不会增加存储空间。如果增加，更新就会失败 。
- 数据库不允许进行删除。使用drop()方法删除collection所有的行。
- 注意: 删除之后，你必须显式的重新创建这个collection。
- 在32bit机器中，capped collection最大存储为1e9(1X109)个字节。

元数据

数据库的信息是存储在集合中。它们使用了系统的命名空间：

```
dbname.system.*
```

在MongoDB数据库中名字空间 <dbname>.system.* 是包含多种系统信息的特殊集合(Collection)，如下:

集合命名空间	描述
dbname.system.namespaces	列出所有名字空间。
dbname.system.indexes	列出所有索引。
dbname.system.profile	包含数据库概要(profile)信息。
dbname.system.users	列出所有可访问数据库的用户。
dbname.local.sources	包含复制对端（slave）的服务器信息和状态。

对于修改系统集合中的对象有如下限制。

在插入数据，可以创建索引。但除此之外该表信息是不可变的(特殊的drop index命令将自动更新相关信息)。

是可修改的。 是可删除的。

MongoDB - 连接

描述

在本教程我们将讨论MongoDB的不同连接方式。

启动 MongoDB服务

在前面的教程中，我们已经讨论了如何启动MongoDB服务，你只需要在MongoDB安装目录的bin目录下执行'mongod'即可。

执行启动操作后，mongodb在输出一些必要信息后不会输出任何信息，之后就等待连接的建立，当连接被建立后，就会开始打印日志信息。

你可以使用MongoDB shell 来连接 MongoDB 服务器。你也可以使用PHP来连接mongodb。本教程我们会使用 MongoDB shell来连接Mongodb服务，之后的章节我们将会介绍如何通过php 来连接MongoDB服务。

默认情况下，MongoDB的启动端口为27017。比MongoDB启动端口大1000的端口为MongoDB的web用户界面，你可以再浏览器中输入 http://localhost:28017 来访问MongoDB的web用户界面。

通过shell连接MongoDB服务

你可以通过执行以下命令来连接MongoDB的服务。

注意：localhost为主机名，这个选项是必须的：

```
mongodb://localhost
```

当你执行以上命令时，你可以看到以下输出结果：

```
D:\mongodb\bin>mongo
MongoDB shell version: 1.8.1
connecting to: test
> mongodb://localhost
...
```

如果你检查从哪里连接到MongoDB的服务器，您可以看到如下信息：

```
D:\mongodb\bin>mongod
mongod --help for help and startup options
Wed Jul 13 14:14:51 [initandlisten] MongoDB starting : pid=2820 port=27017 dbpat
h=/data/db/ 32-bit

** NOTE: when using MongoDB 32 bit, you are limited to about 2 gigabytes of data

**       see http://blog.mongodb.org/post/137788967/32-bit-limitations
**       with --dur, the limit is lower

Wed Jul 13 14:14:51 [initandlisten] db version v1.8.1, pdfile version 4.5
Wed Jul 13 14:14:51 [initandlisten] git version: a429cd4f535b2499cc4130b06ff7c26
f41c00f04
Wed Jul 13 14:14:51 [initandlisten] build sys info: windows (5, 1, 2600, 2, 'Ser
vice Pack 3') BOOST_LIB_VERSION=1_35
Wed Jul 13 14:14:51 [initandlisten] waiting for connections on port 27017
Wed Jul 13 14:14:51 [websvr] web admin interface listening on port 28017
Wed Jul 13 14:15:52 [initandlisten] connection accepted from 127.0.0.1:53272 #1
```

最后一行（标记处），打印了你成功连接上MongoDB服务的信息。

MongoDB连接命令格式

使用用户名和密码连接到MongoDB服务器，你必须使用 'username:password@hostname/dbname' 格式，'username'为用户名，'password' 为密码。

使用用户名和密码连接登陆到默认数据库：

```
mongodb://mongo_admin:AxB6_w3r@localhost/
```

以上命令中，用户 mongo_admin使用密码AxB6_w3r连接到本地的MongoDB服务上。输出结果如下所示：

```
D:\mongodb\bin>mongo
MongoDB shell version: 1.8.1
connecting to: test
> mongodb://mongo_admin:AxB6_w3r@localhost/
...
```

使用用户名和密码连接登陆到指定数据库：

连接到指定数据库的格式如下：

```
mongodb://mongo_admin:AxB6_w3r@localhost/w3r
```

更多连接实例

连接本地数据库服务器，端口是默认的。

```
mongodb://localhost
```

使用用户名fred，密码foobar登录localhost的admin数据库。

```
mongodb://fred:foobar@localhost
```

使用用户名fred，密码foobar登录localhost的baz数据库。

```
mongodb://fred:foobar@localhost/baz
```

连接 replica pair, 服务器1为example1.com服务器2为example2。

```
mongodb://example1.com:27017,example2.com:27017
```

连接 replica set 三台服务器 (端口 27017, 27018, 和27019):

```
mongodb://localhost,localhost:27018,localhost:27019
```

连接 replica set 三台服务器, 写入操作应用在主服务器 并且分布查询到从服务器。

```
mongodb://host1,host2,host3/?slaveOk=true
```

直接连接第一个服务器，无论是replica set一部分或者主服务器或者从服务器。

```
mongodb://host1,host2,host3/?connect=direct;slaveOk=true
```

当你的连接服务器有优先级，还需要列出所有服务器，你可以使用上述连接方式。

安全模式连接到localhost:

```
mongodb://localhost/?safe=true
```

以安全模式连接到replica set，并且等待至少两个复制服务器成功写入，超时时间设置为2秒。

```
mongodb://host1,host2,host3/?safe=true;w=2;wtimeoutMS=2000
```

参数选项说明

标准格式：

```
mongodb://[username:password@]host1[:port1][,host2[:port2],...[,hos
```

标准的连接格式包含了多个选项(options)，如下所示：

选项	描述
replicaSet=name	验证replica set的名称。 Impliesconnect=replicaSet.
slaveOk=true|false	true:在connect=direct模式下，驱动会连接第一台机器，即使这台服务器不是主。在connect=replicaSet模式下，驱动会发送所有的写请求到主并且把读取操作分布在其他从服务器。 false: 在 connect=direct模式下，驱动会自动找寻主服务器. 在connect=replicaSet 模式下，驱动仅仅连接主服务器，并且所有的读写命令都连接到主服务器。
safe=true|ffalse	true: 在执行更新操作之后，驱动都会发送getLastError命令来确保更新成功。(还要参考wtimeoutMS). false: 在每次更新之后，驱动不会发送getLastError来确保更新成功。
w=n	驱动添加 { w : n } 到getLastError命令. 应用于safe=true。
wtimeoutMS=ms	驱动添加 { wtimeout : ms } 到 getlasterror 命令. 应用于 safe=true.
fsync=true|ffalse	true: 驱动添加 { fsync : true } 到 getlasterror 命令.应用于 safe=true. false: 驱动不会添加到getLastError命令中。
journal=true|ffalse	如果设置wie true, 同步到 journal (在提交到数据库前写入到实体中). 应用于 safe=true
connectTimeoutMS=ms	可以打开连接的时间。
socketTimeoutMS=ms	发送和接受sockets的时间。

PHP安装MongoDB扩展驱动

描述

本教程将向大家介绍如何在Linux、window、Mac平台上安装MongoDB扩展。

Linux上安装 MongoDB PHP扩展

在终端上安装

你可以在linux中执行以下命令来安装MongoDB 的 PHP 扩展驱动

```
$ sudo pecl install mongo
```

使用php的pecl安装命令必须保证网络连接可用以及root权限。

安装手册

如果你想通过源码来编译扩展驱动。你必须手动编译源码包，这样做的好是最新修正的bug包含在源码包中。

你可以在Github上下载MongoDB PHP驱动包。访问github网站然后搜索"mongo php driver"(下载地址：https://github.com/mongodb/mongo-php-driver)，下载该源码包，然后执行以下命令：

```
$ tar zxvf mongodb-mongodb-php-driver-<commit_id>.tar.gz
$ cd mongodb-mongodb-php-driver-<commit_id>
$ phpize
$ ./configure
$ sudo make install
```

如果你的php是自己编译的，则安装方法如下(假设是编译在/usr/local/php目录中)：

```
$ tar zxvf mongodb-mongodb-php-driver-<commit_id>.tar.gz
$ cd mongodb-mongodb-php-driver-<commit_id>
$ /usr/local/php/bin/phpize
$ ./configure --with-php-config=/usr/local/php/bin/php-config
$ sudo make install
```

执行以上命令后，你需要修改php.ini文件，在php.ini文件中添加mongo配置，配置如下：

```
extension=mongo.so
```

注意：你需要指明 extension_dir 配置项的路径。

window上安装 MongoDB PHP扩展

Github上已经提供了用于window平台的预编译php mongodb驱动二进制包(下载地址： https://s3.amazonaws.com/drivers.mongodb.org/php/index.html)，你可以下载与你php对应的版本，但是你需要注意以下几点问题：

- VC6 是运行于 Apache 服务器
- 'Thread safe'（线程安全）是运行在Apache上以模块的PHP上，如果你以CGI的模式运行PHP，请选择非线程安全模式（' non-thread safe'）。
- VC9是运行于 IIS 服务器上。
- 下载完你需要的二进制包后，解压压缩包，将'php_mongo.dll'文件添加到你的PHP扩展目录中（ext）。ext目录通常在PHP安装目录下的ext目录。

打开php配置文件 php.ini 添加以下配置：

```
extension=php_mongo.dll
```

重启服务器。

通过浏览器访问phpinfo，如果安装成功，就会看到类型以下的信息：

mongo

MongoDB Support	enabled
Version	1.2.0-

Directive	Local Value	Master Value
mongo.allow_empty_keys	0	0
mongo.allow_persistent	1	1
mongo.auto_reconnect	1	1
mongo.chunk_size	262144	262144
mongo.cmd	$	$
mongo.default_host	localhost	localhost
mongo.default_port	27017	27017
mongo.long_as_object	0	0
mongo.native_long	0	0
mongo.no_id	0	0
mongo.utf8	1	1

MAC中安装MongoDB PHP扩展驱动

你可以使用'autoconf'安装MongoDB PHP扩展驱动。

你可以使用'Xcode'安装MongoDB PHP扩展驱动。

如果你使用 XAMPP，你可以使用以下命令安装MongoDB PHP扩展驱动：

```
sudo /Applications/XAMPP/xamppfiles/bin/pecl install mongo
```

如果以上命令在XMPP或者MAMP中不起作用，你需要在Github上下载兼容的预编译包。

然后添加 'extension=mongo.so'配置到你的php.ini文件中。

MongoDB 数据插入

描述

本章节中我们将向大家介绍如何将数据插入到MongoDB的集合中。

文档的数据结构和JSON基本一样。

所有存储在集合中的数据都是BSON格式。

BSON是一种类json的一种二进制形式的存储格式,简称Binary JSON

。

MongoDB数据库切换

以下命令可以使用"myinfo"数据库：

```
use myinfo switch to db myinfo
```

```
> use myinfo
switched to db myinfo
>
```

为MongoDB数据库定义一个文档

以下文档可以存储在MongoDB中：

```
document=({"user_id" : "ABCDBWN","password" :"ABCDBWN" ,"date_of_jo
"15/10/2010" ,"education" :"B.C.A." , "profession" : "DEVELOPER","i
"MUSIC","community_name" :["MODERN MUSIC", "CLASSICAL
MUSIC","WESTERN MUSIC"],"community_moder_id" : ["MR. BBB","MR. JJJ"
MMM"],"community_members" : [500,200,1500],"friends_id" :
["MMM123","NNN123","OOO123"],"ban_friends_id" :
["BAN123","BAN456","BAN789"]});
```

命令执行如下图所示：

```
> document=({"user_id" : "ABCDBWN","password" :"ABCDBWN" ,"date_of_join" : "15/1
0/2010" ,"education" :"B.C.A." , "profession" : "DEVELOPER","interest" : "MUSIC"
,"community_name" :["MODERN MUSIC", "CLASSICAL MUSIC","WESTERN MUSIC"],"communit
y_moder_id" : ["MR. BBB","MR. JJJ","MR MMM"],"community_members" : [500,200,1500
],"friends_id" : ["MMM123","NNN123","OOO123"],"ban_friends_id" :["BAN123","BAN45
6","BAN789"]});
```

显示已定义的文档

已定义的文档显示格式如下所示：

```
{
        "user_id" : "ABCDBWN",
        "password" : "ABCDBWN",
        "date_of_join" : "15/10/2010",
        "education" : "B.C.A.",
        "profession" : "DEVELOPER",
        "interest" : "MUSIC",
        "community_name" : [
                "MODERN MUSIC",
                "CLASSICAL MUSIC",
                "WESTERN MUSIC"
        ],
        "community_moder_id" : [
                "MR. BBB",
                "MR. JJJ",
                "MR MMM"
        ],
        "community_members" : [
                500,
                200,
                1500
        ],
        "friends_id" : [
                "MMM123",
                "NNN123",
                "OOO123"
        ],
        "ban_friends_id" : [
                "BAN123",
                "BAN456",
                "BAN789"
        ]
}
```

在集合中插入文档

将以上的文档数据存储到"myinfo" 数据库中的 "userdetails" 集合，执行如下命令：

```
db.userdetails.insert(document)
```

```
>
> db.userdetails.insert(document);
> _
```

使用换行符插入数据

当文档的数据较多的时候，我们可以使用换行符来分割文档数据，如下所示：

```
document=({"user_id" : "ABCDBWN","password" :"ABCDBWN" ,"date_of_jo
"education" :"B.C.A." , "profession" : "DEVELOPER","interest" : "MU
"community_name" :["MODERN MUSIC", "CLASSICAL MUSIC","WESTERN MUSIC
"community_moder_id" : ["MR. BBB","MR. JJJ","MR MMM"],
"community_members" : [500,200,1500],"friends_id" : ["MMM123","NNN1
"ban_friends_id" :["BAN123","BAN456","BAN789"]});
```

命令执行如下图所示：

```
> document=({"user_id" : "ABCDBWN","password" :"ABCDBWN" ,"date_of_join" : "15/1
0/2010" ,
... "education" :"B.C.A." , "profession" : "DEVELOPER","interest" : "MUSIC",
... "community_name" :["MODERN MUSIC", "CLASSICAL MUSIC","WESTERN MUSIC"],
... "community_moder_id" : ["MR. BBB","MR. JJJ","MR MMM"],
... "community_members" : [500,200,1500],"friends_id" : ["MMM123","NNN123","OOO1
23"],
... "ban_friends_id" :["BAN123","BAN456","BAN789"]});
```

集合中直接插入数据（无定义文档）

数据可以不用定义文档通过shell直接插入：

```
db.userdetails.insert({"user_id" : "xyz123","password" :"xyz123" ,'
"education" :"M.C.A." , "profession" : "Software consultant","inter
"community" : [
{
"name" : "DDD FILM CLUB",
"moder_id" : "MR. DBNA",
"members" : "25000",
},
{
"name" : "AXN MOVIES",
"moder_id" : "DOGLUS HUNT",
"members" : "15000",
},
{
"name" : "UROPEAN FILM LOVERS",
"moder_id" : "AMANT LUIS",
"members" : "20000",
}
],
"friends" :[
{
"user_id" : "KKK258",
},
{
"user_id" : "LLL147",
},
{
"user_id" : "MMM369",
}
],
"ban_friends" :[
{
"user_id" : "BAN147"
},
{
"user_id" : "BAN258"
},
{
"user_id" : "BAN369"
}
]
});
```

命令执行如下图所示：

```
>
> db.userdetails.insert({"user_id" : "xyz123","password" :"xyz123" ,"date_of_joi
n" : "15/08/2010" ,
... "education" :"M.C.A." , "profession" : "Software consultant","interest" : "F
ilm",
... "community" : [
... {
... "name" : "DDD FILM CLUB",
... "moder_id" : "MR. DBNA",
... "members" : "25000",
... },
... {
... "name" : "AXN MOVIES",
... "moder_id" : "DOGLUS HUNT",
... "members" : "15000",
... },
... {
... "name" : "UROPEAN FILM LOVERS",
... "moder_id" : "AMANT LUIS",
... "members" : "20000",
... }
... ],
... "friends" :[
... {
... "user_id" : "KKK258",
... },
... {
... "user_id" : "LLL147",
... },
... {
... "user_id" : "MMM369",
... }
... ],
... "ban_friends" :[
... {
... "user_id" : "BAN147"
... },
... {
... "user_id" : "BAN258"
... },
... {
... "user_id" : "BAN369"
... }
... ]
... });
>
>
```

查看集合中的数据

使用以下命令查看集合中的数据：

```
db.userdetails.find();
```

```
{
        "user_id" : "xyz123",
        "password" : "xyz123",
        "date_of_join" : "15/08/2010",
        "education" : "M.C.A.",
        "profession" : "Software consultant",
        "interest" : "Film",
        "community" : [
                {
                        "name" : "DDD FILM CLUB",
                        "moder_id" : "MR. DBNA",
                        "members" : "25000"
                },
                {
                        "name" : "AXN MOVIES",
                        "moder_id" : "DOGLUS HUNT",
                        "members" : "15000"
                },
                {
                        "name" : "UROPEAN FILM LOVERS",
                        "moder_id" : "AMANT LUIS",
                        "members" : "20000"
                }
        ],
        "friends" : [
                {
                        "user_id" : "KKK258"
                },
                {
                        "user_id" : "LLL147"
                },
                {
                        "user_id" : "MMM369"
                }
        ],
        "ban_friends" : [
                {
                        "user_id" : "BAN147"
                },
                {
                        "user_id" : "BAN258"
                },
                {
                        "user_id" : "BAN369"
                }
        ]
}
```

MongoDB使用update()函数更新数据

描述

本章节我们将开始学习如何更新MongoDB中的集合数据。

MongoDB数据更新可以使用update()函数。

```
db.collection.update( criteria, objNew, upsert, multi )
```

update()函数接受以下四个参数：

- **criteria** : update的查询条件，类似sql update查询内where后面的。
- **objNew** : update的对象和一些更新的操作符（如$,$inc...）等，也可以理解为sql update查询内set后面的
- **upsert** : 这个参数的意思是，如果不存在update的记录，是否插入objNew,true为插入，默认是false，不插入。
- **multi** : mongodb默认是false,只更新找到的第一条记录，如果这个参数为true,就把按条件查出来多条记录全部更新。

在本教程中我们使用的数据库名称为"myinfo"，集合名称为"userdetails"，以下为插入的数据：

```
> document=({"user_id" : "MNOPBWN","password" :"MNOPBWN" ,"date_of_
,"education" :"M.C.A." , "profession" : "CONSULTANT","interest" : '
"CLASSICAL MUSIC","WESTERN MUSIC"],"community_moder_id" : ["MR. BBB
[500,200,1500],"friends_id" : ["MMM123","NNN123","OOO123"],"ban_fri
```

```
> db.userdetails.insert(document)
```

```
> document=({"user_id" : "QRSTBWN","password" :"QRSTBWN" ,"date_of_
, "profession" : "MARKETING","interest" : "MUSIC","community_name"
MUSIC"],"community_moder_id" : ["MR. BBB","MR. JJJ","MR MMM"],"comm
["MMM123","NNN123","OOO123"],"ban_friends_id" :["BAN123","BAN456",'
```

```
> db.userdetails.insert(document)
```

update() 命令

如果我们想将"userdetails"集合中"user_id"为"QRSTBWN"的"password"字段修改为"NEWPASSWORD"，那么我们可以使用update()命令来实现（如下实例所示）。

如果criteria参数匹配集合中的任何一条数据，它将会执行替换命令，否则会插入一条新的数据。

以下实例将更新第一条匹配条件的数据：

```
> db.userdetails.update({"user_id" : "QRSTBWN"},{"user_id" : "QRSTE
,"date_of_join" : "17/10/2010" ,"education" :"M.B.A." , "professior
"MUSIC","community_name" :["MODERN MUSIC", "CLASSICAL MUSIC","WESTE
BBB","MR. JJJ","MR MMM"],"community_members" : [500,200,1500],"frie
```

```
>
> db.userdetails.update({"user_id" : "QRSTBWN"},{"user_id" : "QRSTBWN","password
" :"NEWPASSWORD" ,"date_of_join" : "17/10/2010" ,"education" :"M.B.A." , "profes
sion" : "MARKETING","interest" : "MUSIC","community_name" :["MODERN MUSIC", "CLA
SSICAL MUSIC","WESTERN MUSIC"],"community_moder_id" : ["MR. BBB","MR. JJJ","MR M
MM"],"community_members" : [500,200,1500],"friends_id" : ["MMM123","NNN123","OOO
123"],"ban_friends_id" :["BAN123","BAN456","BAN789"]});
>
>
```

查看集合中更新后的数据

我们可以使用以下命令查看数据是否更新：

```
> db.userdetails.find();
```

```
] }
{ "_id" : ObjectId("4e26ca5f54e3b76638e4dbf6"), "user_id" : "MNOPBWN", "password
" : "MNOPBWN", "date_of_join" : "16/10/2010", "education" : "M.C.A.", "professio
n" : "CONSULTANT", "interest" : "MUSIC", "community_name" : [ "MODERN MUSIC", "C
LASSICAL MUSIC", "WESTERN MUSIC" ], "community_moder_id" : [ "MR. BBB", "MR. JJJ
", "MR MMM" ], "community_members" : [ 500, 200, 1500 ], "friends_id" : [ "MMM12
3", "NNN123", "OOO123" ], "ban_friends_id" : [ "BAN123", "BAN456", "BAN789" ] }
{ "_id" : ObjectId("4e26ca8754e3b76638e4dbf7"), "user_id" : "QRSTBWN", "password
" : "NEWPASSWORD", "date_of_join" : "17/10/2010", "education" : "M.B.A.", "profe
ssion" : "MARKETING", "interest" : "MUSIC", "community_name" : [ "MODERN MUSIC",
 "CLASSICAL MUSIC", "WESTERN MUSIC" ], "community_moder_id" : [ "MR. BBB", "MR.
JJJ", "MR MMM" ], "community_members" : [ 500, 200, 1500 ], "friends_id" : [ "MM
M123", "NNN123", "OOO123" ], "ban_friends_id" : [ "BAN123", "BAN456", "BAN789" ]
 }
>
>
```

更多实例

只更新第一条记录：

```
db.test0.update( { "count" : { $gt : 1 } } , { $set : { "test2" : '
```

全部更新：

```
db.test0.update( { "count" : { $gt : 3 } } , { $set : { "test2" : '
```

只添加第一条：

```
db.test0.update( { "count" : { $gt : 4 } } , { $set : { "test5" : '
```

全部添加加进去:

```
db.test0.update( { "count" : { $gt : 5 } } , { $set : { "test5" : '
```

全部更新：

```
db.test0.update( { "count" : { $gt : 15 } } , { $inc : { "count" :
```

只更新第一条记录：

```
db.test0.update( { "count" : { $gt : 10 } } , { $inc : { "count" :
```

MongoDB使用- remove()函数删除数据

描述

在前面的几个章节中我们已经学习了MongoDB中如何为集合添加数据和更新数据。在本章节中我们将继续学习MongoDB集合的删除。

MongoDB remove()函数是用来移除集合中的数据。

MongoDB数据更新可以使用update()函数。在执行remove()函数前先执行find()命令来判断执行的条件是否正确，这是一个比较好的习惯。

我们使用的数据库名称为**"myinfo"** 我们的集合名称为**"userdetails"**，以下为我们插入的数据：

```
> document=({"user_id" : "testuser","password" :"testpassword" ,"da
:"M.C.A." , "profession" : "CONSULTANT","interest" : "MUSIC","commu
MUSIC","WESTERN MUSIC"],"community_moder_id" : ["MR. BBB","MR. JJJ"
[500,200,1500],"friends_id" : ["MMM123","NNN123","OOO123"],"ban_fri
```

```
> db.userdetails.insert(document)
```

查看集合中已经插入的数据

```
> db.userdetails.find();
```

```
        }
] }
{ "_id" : ObjectId("4e26ca5f54e3b76638e4dbf6"), "user_id" : "MNOPBWN", "password
" : "MNOPBWN", "date_of_join" : "16/10/2010", "education" : "M.C.A.", "professio
n" : "CONSULTANT", "interest" : "MUSIC", "community_name" : [ "MODERN MUSIC", "C
LASSICAL MUSIC", "WESTERN MUSIC" ], "community_moder_id" : [ "MR. BBB", "MR. JJJ
", "MR MMM" ], "community_members" : [ 500, 200, 1500 ], "friends_id" : [ "MMM12
3", "NNN123", "OOO123" ], "ban_friends_id" : [ "BAN123", "BAN456", "BAN789" ] }
{ "_id" : ObjectId("4e26ca8754e3b76638e4dbf7"), "user_id" : "QRSTBWN", "password
" : "NEWPASSWORD", "date_of_join" : "17/10/2010", "education" : "M.B.A.", "profe
ssion" : "MARKETING", "interest" : "MUSIC", "community_name" : [ "MODERN MUSIC",
 "CLASSICAL MUSIC", "WESTERN MUSIC" ], "community_moder_id" : [ "MR. BBB", "MR.
JJJ", "MR MMM" ], "community_members" : [ 500, 200, 1500 ], "friends_id" : [ "MM
M123", "NNN123", "OOO123" ], "ban_friends_id" : [ "BAN123", "BAN456", "BAN789" ]
 }
{ "_id" : ObjectId("4e327b4bb94947d1cab92065"), "user_id" : "testuser", "passwor
d" : "testpassword", "date_of_join" : "16/10/2010", "education" : "M.C.A.", "pro
fession" : "CONSULTANT", "interest" : "MUSIC", "community_name" : [ "MODERN MUSI
C", "CLASSICAL MUSIC", "WESTERN MUSIC" ], "community_moder_id" : [ "MR. BBB", "M
R. JJJ", "MR MMM" ], "community_members" : [ 500, 200, 1500 ], "friends_id" : [
"MMM123", "NNN123", "OOO123" ], "ban_friends_id" : [ "BAN123", "BAN456", "BAN789
" ] }
>
```

使用 remove() 函数移除数据

如果你想移除"userdetails"集合中"user_id" 为 "testuser"的数据你可以执行以下命令：

```
> db.userdetails.remove( { "user_id" : "testuser" } )
```

删除所有数据

如果你想删除"userdetails"集合中的所有数据，可以执行以下命令：

```
> db.userdetails.remove({})
```

使用drop()删除集合

如果你想删除整个"userdetails"集合，包含所有文档数据，可以执行以下数据：

```
> db.userdetails.drop()
```

```
>
> db.userdetails.drop()
true
>
>
> _
```

drop()函数返回 true或者false。以上执行结果返回了true，说明操作成功。

使用dropDatabase()函数删除数据库

如果你想删除整个数据库的数据，你可以执行以下命令：

```
> db.dropDatabase()
```

执行命令前查看当前使用的数据库是一个良好的习惯，这样可以确保你要删除数据库是正确的，以免造成误操作而产生数据丢失的后果：

```
>
> db
myinfo
>
>
```

```
>
> db.dropDatabase()
{ "dropped" : "myinfo", "ok" : 1 }
> _
```

MongoDB 查询

描述

本教程我们将向大家介绍如何在MongoDB集合中获取数据。

我们使用的数据库名称为**"myinfo"** 我们的集合名称为**"userdetails"**，以下为我们插入的数据：

```
> db.userdetails.insert({"user_id" : "user1","password" :"1a2b3c" ,
, "profession" : "CONSULTANT","interest" : "MUSIC","community_name"
MUSIC"],"community_moder_id" : ["MR. Alex","MR. Dang","MR Haris"],"
[700,200,1500],"friends_id" : ["kumar","harry","anand"],"ban_frienc
```

```
> db.userdetails.insert({"user_id" : "user2","password" :"11aa1a" ,
:"M.B.A." , "profession" : "MARKETING","interest" : "MUSIC","commur
MUSIC","WESTERN MUSIC"],"community_moder_id" : ["MR. Roy","MR. Das"
[500,300,1400],"friends_id" : ["pal","viki","john"],"ban_friends_ic
```

```
> db.userdetails.insert({"user_id" : "user3","password" :"b1c1d1" ,
:"M.C.A." , "profession" : "IT COR.","interest" : "ART","community_
ART"],"community_moder_id" : ["MR. Rifel","MR. Sarma","MR Bhatia"],
[5000,2000,1500],"friends_id" : ["philip","anant","alan"],"ban_frie
```

```
> db.userdetails.insert({"user_id" : "user4","password" :"abczyx" ,
:"M.B.B.S." , "profession" : "DOCTOR","interest" : "SPORTS","commur
GYES","FAVOURIT GAMES"],"community_moder_id" : ["MR. Paul","MR. Das
[2500,2200,3500],"friends_id" : ["vinod","viki","john"],"ban_frienc
```

从集合中获取数据

如果你想在集合中读取所有的的数据，可以执行以下命令

```
> db.userdetails.find();
```

类似于如下SQL查询语句：

```
Select * from userdetails;
```

输出数据如下所示：

```
>
> db.userdetails.find()
{ "_id" : ObjectId("4e367fd6449f200f44a29f00"), "user_id" : "user1", "password"
: "1a2b3c", "date_of_join" : "16/10/2010", "education" : "M.C.A.", "profession"
: "CONSULTANT", "interest" : "MUSIC", "community_name" : [ "MODERN MUSIC", "CLAS
SICAL MUSIC", "WESTERN MUSIC" ], "community_moder_id" : [ "MR. Alex", "MR. Dang"
, "MR Haris" ], "community_members" : [ 700, 200, 1500 ], "friends_id" : [ "kuma
r", "harry", "anand" ], "ban_friends_id" : [ "Amir", "Raja", "mont" ] }
{ "_id" : ObjectId("4e367feb449f200f44a29f01"), "user_id" : "user2", "password"
: "11aa1a", "date_of_join" : "17/10/2009", "education" : "M.B.A.", "profession"
: "MARKETING", "interest" : "MUSIC", "community_name" : [ "MODERN MUSIC", "CLASS
ICAL MUSIC", "WESTERN MUSIC" ], "community_moder_id" : [ "MR. Roy", "MR. Das", "
MR Doglus" ], "community_members" : [ 500, 300, 1400 ], "friends_id" : [ "pal",
"viki", "john" ], "ban_friends_id" : [ "jalan", "monoj", "evan" ] }
{ "_id" : ObjectId("4e367ff7449f200f44a29f02"), "user_id" : "user3", "password"
: "b1c1d1", "date_of_join" : "16/10/2010", "education" : "M.C.A.", "profession"
: "IT COR.", "interest" : "ART", "community_name" : [ "MODERN ART", "CLASSICAL A
RT", "WESTERN ART" ], "community_moder_id" : [ "MR. Rifel", "MR. Sarma", "MR Bha
tia" ], "community_members" : [ 5000, 2000, 1500 ], "friends_id" : [ "philip", "
anant", "alan" ], "ban_friends_id" : [ "Amir", "Raja", "mont" ] }
{ "_id" : ObjectId("4e368006449f200f44a29f03"), "user_id" : "user4", "password"
: "abczyx", "date_of_join" : "17/8/2009", "education" : "M.B.B.S.", "profession"
 : "DOCTOR", "interest" : "SPORTS", "community_name" : [ "ATHELATIC", "GAMES FAN
GYES", "FAVOURIT GAMES" ], "community_moder_id" : [ "MR. Paul", "MR. Das", "MR
Doglus" ], "community_members" : [ 2500, 2200, 3500 ], "friends_id" : [ "vinod",
 "viki", "john" ], "ban_friends_id" : [ "jalan", "monoj", "evan" ] }
>
```

通过指定条件读取数据

如果我们想在集合"userdetails"中读取"education"为"M.C.A." 的数据，我们可以执行以下命令：

```
> db.userdetails.find({"education":"M.C.A."})
```

类似如下SQL查询语句:

```
Select * from userdetails where education="M.C.A.";
```

输出结果如下所示：

```
>
> db.userdetails.find({"education":"M.C.A."})
{ "_id" : ObjectId("4e367fd6449f200f44a29f00"), "user_id" : "user1", "password"
: "1a2b3c", "date_of_join" : "16/10/2010", "education" : "M.C.A.", "profession"
: "CONSULTANT", "interest" : "MUSIC", "community_name" : [ "MODERN MUSIC", "CLAS
SICAL MUSIC", "WESTERN MUSIC" ], "community_moder_id" : [ "MR. Alex", "MR. Dang"
, "MR Haris" ], "community_members" : [ 700, 200, 1500 ], "friends_id" : [ "kuma
r", "harry", "anand" ], "ban_friends_id" : [ "Amir", "Raja", "mont" ] }
{ "_id" : ObjectId("4e367ff7449f200f44a29f02"), "user_id" : "user3", "password"
: "b1c1d1", "date_of_join" : "16/10/2010", "education" : "M.C.A.", "profession"
: "IT COR.", "interest" : "ART", "community_name" : [ "MODERN ART", "CLASSICAL A
RT", "WESTERN ART" ], "community_moder_id" : [ "MR. Rifel", "MR. Sarma", "MR Bha
tia" ], "community_members" : [ 5000, 2000, 1500 ], "friends_id" : [ "philip", "
anant", "alan" ], "ban_friends_id" : [ "Amir", "Raja", "mont" ] }
> _
```

MongoDB条件操作符

描述

条件操作符用于比较两个表达式并从mongoDB集合中获取数据。

在本章节中，我们将讨论如何在MongoDB中使用条件操作符。

MongoDB中条件操作符有：

- (>) 大于 - $gt
- (<) 小于 - $lt
- (>=) 大于等于 - $gte
- (<=) 小于等于 - $lte

我们使用的数据库名称为**"myinfo"** 我们的集合名称为**"testtable"**，以下为我们插入的数据。

简单的集合"testtable"：

```
>
> db.testtable.find()
{ "_id" : ObjectId("4e3d1ee9f52b9bd78374f952"), "user_id" : "user1", "password"
: "1a2b3c", "sex" : "Male", "age" : 17, "date_of_join" : "16/10/2010", "educatio
n" : "M.C.A.", "profession" : "CONSULTANT", "interest" : "MUSIC", "extra" : { "c
ommunity_name" : [ "MODERN MUSIC", "CLASSICAL MUSIC", "WESTERN MUSIC" ], "commun
ity_moder_id" : [ "MR. Alex", "MR. Dang", "MR Haris" ], "community_members" : [
700, 200, 1500 ], "friends" : { "valued_friends_id" : [ "kumar", "harry", "anand
" ], "ban_friends_id" : [ "Amir", "Raja", "mont" ] } } }
{ "_id" : ObjectId("4e3d1f1cf52b9bd78374f953"), "user_id" : "user2", "password"
: "11aa1a", "sex" : "Male", "age" : 24, "date_of_join" : "17/10/2009", "educatio
n" : "M.B.A.", "profession" : "MARKETING", "interest" : "MUSIC", "extra" : { "co
mmunity_name" : [ "MODERN MUSIC", "CLASSICAL MUSIC", "WESTERN MUSIC" ], "communi
ty_moder_id" : [ "MR. Roy", "MR. Das", "MR Doglus" ], "community_members" : [ 50
0, 300, 1400 ], "friends" : { "valuled_friends_id" : [ "pal", "viki", "john" ],
"ban_friends_id" : [ "jalan", "monoj", "evan" ] } } }
{ "_id" : ObjectId("4e3d1f2af52b9bd78374f954"), "user_id" : "user3", "password"
: "b1c1d1", "sex" : "Female", "age" : 19, "date_of_join" : "16/10/2010", "educat
ion" : "M.C.A.", "profession" : "IT COR.", "interest" : "ART", "extra" : { "comm
unity_name" : [ "MODERN ART", "CLASSICAL ART", "WESTERN ART" ], "community_moder
_id" : [ "MR. Rifel", "MR. Sarma", "MR Bhatia" ], "community_members" : [ 5000,
2000, 1500 ], "friends" : { "valued_friends_id" : [ "philip", "anant", "alan" ],
 "ban_friends_id" : [ "Amir", "Raja", "mont" ] } } }
{ "_id" : ObjectId("4e3d1f35f52b9bd78374f955"), "user_id" : "user4", "password"
: "abczyx", "sex" : "Female", "age" : 22, "date_of_join" : "17/8/2009", "educati
on" : "M.B.B.S.", "profession" : "DOCTOR", "interest" : "SPORTS", "extra" : { "c
ommunity_name" : [ "ATHELATIC", "GAMES FAN GYES", "FAVOURIT GAMES" ], "community
_moder_id" : [ "MR. Paul", "MR. Das", "MR Doglus" ], "community_members" : [ 250
0, 2200, 3500 ], "friends" : { "valued_friends_id" : [ "vinod", "viki", "john" ]
, "ban_friends_id" : [ "jalan", "monoj", "evan" ] } } }
>
>
```

MongoDB (>) 大于操作符 - $gt

如果你想获取"testtable"集合中"age" 大于22的数据，你可以使用以下命令：

```
> db.testtable.find({age : {$gt : 22}})
```

类似于SQL语句：

```
Select * from testtable where age >22;
```

输出结果：

```
>
> db.testtable.find({age : {$gt : 22}})
{ "_id" : ObjectId("4e3d1f1cf52b9bd78374f953"), "user_id" : "user2", "password"
: "11aa1a", "sex" : "Male", "age" : 24, "date_of_join" : "17/10/2009", "educatio
n" : "M.B.A.", "profession" : "MARKETING", "interest" : "MUSIC", "extra" : { "co
mmunity_name" : [ "MODERN MUSIC", "CLASSICAL MUSIC", "WESTERN MUSIC" ], "communi
ty_moder_id" : [ "MR. Roy", "MR. Das", "MR Doglus" ], "community_members" : [ 50
0, 300, 1400 ], "friends" : { "valuled_friends_id" : [ "pal", "viki", "john" ],
"ban_friends_id" : [ "jalan", "monoj", "evan" ] } } }
>
> _
```

MongoDB（>=）大于等于操作符 - $gte

如果你想获取"testtable"集合中"age" 大于等于22的数据，你可以执行以下命令:

```
> db.testtable.find({age : {$gte : 22}})
```

类似于SQL语句：

```
Select * from testtable where age >=22;
```

输出结果：

```
>
> db.testtable.find({age : {$gte : 22}})
{ "_id" : ObjectId("4e3d1f1cf52b9bd78374f953"), "user_id" : "user2", "password"
: "11aa1a", "sex" : "Male", "age" : 24, "date_of_join" : "17/10/2009", "educatio
n" : "M.B.A.", "profession" : "MARKETING", "interest" : "MUSIC", "extra" : { "co
mmunity_name" : [ "MODERN MUSIC", "CLASSICAL MUSIC", "WESTERN MUSIC" ], "communi
ty_moder_id" : [ "MR. Roy", "MR. Das", "MR Doglus" ], "community_members" : [ 50
0, 300, 1400 ], "friends" : { "valuled_friends_id" : [ "pal", "viki", "john" ],
"ban_friends_id" : [ "jalan", "monoj", "evan" ] } } }
{ "_id" : ObjectId("4e3d1f35f52b9bd78374f955"), "user_id" : "user4", "password"
: "abczyx", "sex" : "Female", "age" : 22, "date_of_join" : "17/8/2009", "educati
on" : "M.B.B.S.", "profession" : "DOCTOR", "interest" : "SPORTS", "extra" : { "c
ommunity_name" : [ "ATHELATIC", "GAMES FAN GYES", "FAVOURIT GAMES" ], "community
_moder_id" : [ "MR. Paul", "MR. Das", "MR Doglus" ], "community_members" : [ 250
0, 2200, 3500 ], "friends" : { "valued_friends_id" : [ "vinod", "viki", "john" ]
, "ban_friends_id" : [ "jalan", "monoj", "evan" ] } } }
>
>
```

MongoDB (<) 小于操作符 - $lt

如果你想获取"testtable"集合中"age" 小于19的数据，你可以执行以下命令：

类似于SQL语句：

```
Select * from testtable where age <19;
```

输出结果：

```
>
> db.testtable.find({age : {$lt :19}})
{ "_id" : ObjectId("4e3d1ee9f52b9bd78374f952"), "user_id" : "user1", "password"
: "1a2b3c", "sex" : "Male", "age" : 17, "date_of_join" : "16/10/2010", "educatio
n" : "M.C.A.", "profession" : "CONSULTANT", "interest" : "MUSIC", "extra" : { "c
ommunity_name" : [ "MODERN MUSIC", "CLASSICAL MUSIC", "WESTERN MUSIC" ], "commun
ity_moder_id" : [ "MR. Alex", "MR. Dang", "MR Haris" ], "community_members" : [
700, 200, 1500 ], "friends" : { "valued_friends_id" : [ "kumar", "harry", "anand
" ], "ban_friends_id" : [ "Amir", "Raja", "mont" ] } } }
>
>
```

MongoDB (<=) 小于操作符 - $lte

如果你想获取"testtable"集合中"age" 小于等于19的数据，你可以执行以下命令：

```
> db.testtable.find({age : {$lte : 19}})
```

类似于SQL语句：

```
Select * from testtable where age <=19;
```

输出结果：

```
>
> db.testtable.find({age : {$lte : 19}})
{ "_id" : ObjectId("4e3d1ee9f52b9bd78374f952"), "user_id" : "user1", "password"
: "1a2b3c", "sex" : "Male", "age" : 17, "date_of_join" : "16/10/2010", "educatio
n" : "M.C.A.", "profession" : "CONSULTANT", "interest" : "MUSIC", "extra" : { "c
ommunity_name" : [ "MODERN MUSIC", "CLASSICAL MUSIC", "WESTERN MUSIC" ], "commun
ity_moder_id" : [ "MR. Alex", "MR. Dang", "MR Haris" ], "community_members" : [
700, 200, 1500 ], "friends" : { "valued_friends_id" : [ "kumar", "harry", "anand
" ], "ban_friends_id" : [ "Amir", "Raja", "mont" ] } } }
{ "_id" : ObjectId("4e3d1f2af52b9bd78374f954"), "user_id" : "user3", "password"
: "b1c1d1", "sex" : "Female", "age" : 19, "date_of_join" : "16/10/2010", "educat
ion" : "M.C.A.", "profession" : "IT COR.", "interest" : "ART", "extra" : { "comm
unity_name" : [ "MODERN ART", "CLASSICAL ART", "WESTERN ART" ], "community_moder
_id" : [ "MR. Rifel", "MR. Sarma", "MR Bhatia" ], "community_members" : [ 5000,
2000, 1500 ], "friends" : { "valued_friends_id" : [ "philip", "anant", "alan" ],
 "ban_friends_id" : [ "Amir", "Raja", "mont" ] } } }
>
>
```

MongoDB 使用 (<) 和 (>) 查询operator - $lt 和 $gt

如果你想获取"testtable"集合中"age" 大于17以及小于24的数据，你可以执行以下命令：

```
> db.testtable.find({age : {$lt :24, $gt : 17}})
```

类似于SQL语句：

```
Select * from testtable where age 17;
```

输出结果：

```
>
> db.testtable.find({age : {$lt :24, $gt : 17}})
{ "_id" : ObjectId("4e3d1f2af52b9bd78374f954"), "user_id" : "user3", "password"
: "b1c1d1", "sex" : "Female", "age" : 19, "date_of_join" : "16/10/2010", "educat
ion" : "M.C.A.", "profession" : "IT COR.", "interest" : "ART", "extra" : { "comm
unity_nane" : [ "MODERN ART", "CLASSICAL ART", "WESTERN ART" ], "community_moder
_id" : [ "MR. Rifel", "MR. Sarma", "MR Bhatia" ], "community_members" : [ 5000,
2000, 1500 ], "friends" : { "valued_friends_id" : [ "philip", "anant", "alan" ],
 "ban_friends_id" : [ "Amir", "Raja", "mont" ] } } }
{ "_id" : ObjectId("4e3d1f35f52b9bd78374f955"), "user_id" : "user4", "password"
: "abczyx", "sex" : "Female", "age" : 22, "date_of_join" : "17/8/2009", "educati
on" : "M.B.B.S.", "profession" : "DOCTOR", "interest" : "SPORTS", "extra" : { "c
ommunity_name" : [ "ATHELATIC", "GAMES FAN GYES", "FAVOURIT GAMES" ], "community
_moder_id" : [ "MR. Paul", "MR. Das", "MR Doglus" ], "community_members" : [ 250
0, 2200, 3500 ], "friends" : { "valued_friends_id" : [ "vinod", "viki", "john" ]
, "ban_friends_id" : [ "jalan", "monoj", "evan" ] } } }
>
> _
```

MongoDB条件操作符 - $type

描述

在本章节中，我们将继续讨论MongoDB中条件操作符 $type。

$type操作符是基于BSON类型来检索集合中匹配的结果。

MongoDB中可以使用的类型：

类型描述	类型值
Double	1
String	2
Object	3
Array	4
Binary data	5
Object id	7
Boolean	8
Date	9
Null	10
Regular expression	11
JavaScript code	13
Symbol	14
JavaScript code with scope	15
32-bit integer	16
Timestamp	17
64-bit integer	18
Min key	255
Max key	127

我们使用的数据库名称为**"myinfo"** 我们的集合名称为**"testtable"**，以下为我们插入的数据。

简单的集合"testtable"：

```
>
> db.testtable.find()
{ "_id" : ObjectId("4e3d1ee9f52b9bd78374f952"), "user_id" : "user1", "password"
: "1a2b3c", "sex" : "Male", "age" : 17, "date_of_join" : "16/10/2010", "educatio
n" : "M.C.A.", "profession" : "CONSULTANT", "interest" : "MUSIC", "extra" : { "c
ommunity_name" : [ "MODERN MUSIC", "CLASSICAL MUSIC", "WESTERN MUSIC" ], "commun
ity_moder_id" : [ "MR. Alex", "MR. Dang", "MR Haris" ], "community_members" : [
700, 200, 1500 ], "friends" : { "valued_friends_id" : [ "kumar", "harry", "anand
" ], "ban_friends_id" : [ "Amir", "Raja", "mont" ] } } }
{ "_id" : ObjectId("4e3d1f1cf52b9bd78374f953"), "user_id" : "user2", "password"
: "11aa1a", "sex" : "Male", "age" : 24, "date_of_join" : "17/10/2009", "educatio
n" : "M.B.A.", "profession" : "MARKETING", "interest" : "MUSIC", "extra" : { "co
mmunity_name" : [ "MODERN MUSIC", "CLASSICAL MUSIC", "WESTERN MUSIC" ], "communi
ty_moder_id" : [ "MR. Roy", "MR. Das", "MR Doglus" ], "community_members" : [ 50
0, 300, 1400 ], "friends" : { "valuled_friends_id" : [ "pal", "viki", "john" ],
"ban_friends_id" : [ "jalan", "monoj", "evan" ] } } }
{ "_id" : ObjectId("4e3d1f2af52b9bd78374f954"), "user_id" : "user3", "password"
: "b1c1d1", "sex" : "Female", "age" : 19, "date_of_join" : "16/10/2010", "educat
ion" : "M.C.A.", "profession" : "IT COR.", "interest" : "ART", "extra" : { "comm
unity_name" : [ "MODERN ART", "CLASSICAL ART", "WESTERN ART" ], "community_moder
_id" : [ "MR. Rifel", "MR. Sarma", "MR Bhatia" ], "community_members" : [ 5000,
2000, 1500 ], "friends" : { "valued_friends_id" : [ "philip", "anant", "alan" ],
 "ban_friends_id" : [ "Amir", "Raja", "mont" ] } } }
{ "_id" : ObjectId("4e3d1f35f52b9bd78374f955"), "user_id" : "user4", "password"
: "abczyx", "sex" : "Female", "age" : 22, "date_of_join" : "17/8/2009", "educati
on" : "M.B.B.S.", "profession" : "DOCTOR", "interest" : "SPORTS", "extra" : { "c
ommunity_name" : [ "ATHELATIC", "GAMES FAN GYES", "FAVOURIT GAMES" ], "community
_moder_id" : [ "MR. Paul", "MR. Das", "MR Doglus" ], "community_members" : [ 250
0, 2200, 3500 ], "friends" : { "valued_friends_id" : [ "vinod", "viki", "john" ]
, "ban_friends_id" : [ "jalan", "monoj", "evan" ] } } }
{ "_id" : ObjectId("4e578544d10a10ad4b593f24"), "user_id" : "user5", "password"
: "user5", "sex" : "Male", "age" : 21, "date_of_join" : "17/08/2011", "education
" : "MCA", "profession" : "S.W. Engineer", "interest" : "SPORTS", "extra" : { "c
ommunity_name" : [ "ATHELATIC", "GAMES FAN GYES", "FAVOURIT GAMES" ] } }
>
>
```

MongoDB 操作符 - $type 实例

如果想获取 "testtable" 集合包含在 "extra" 中的"friends"为BSON类型的对象，你可以使用以下命令：

```
> db.testtable.find({"extra.friends" : {$type : 3}})
```

```
>
> db.testtable.find({"extra.friends" : {$type : 3}})
{ "_id" : ObjectId("4e3d1ee9f52b9bd78374f952"), "user_id" : "user1", "password"
: "1a2b3c", "sex" : "Male", "age" : 17, "date_of_join" : "16/10/2010", "educatio
n" : "M.C.A.", "profession" : "CONSULTANT", "interest" : "MUSIC", "extra" : { "c
ommunity_name" : [ "MODERN MUSIC", "CLASSICAL MUSIC", "WESTERN MUSIC" ], "commun
ity_moder_id" : [ "MR. Alex", "MR. Dang", "MR Haris" ], "community_members" : [
700, 200, 1500 ], "friends" : { "valued_friends_id" : [ "kumar", "harry", "anand
" ], "ban_friends_id" : [ "Amir", "Raja", "mont" ] } } }
{ "_id" : ObjectId("4e3d1f1cf52b9bd78374f953"), "user_id" : "user2", "password"
: "11aa1a", "sex" : "Male", "age" : 24, "date_of_join" : "17/10/2009", "educatio
n" : "M.B.A.", "profession" : "MARKETING", "interest" : "MUSIC", "extra" : { "co
mmunity_name" : [ "MODERN MUSIC", "CLASSICAL MUSIC", "WESTERN MUSIC" ], "communi
ty_moder_id" : [ "MR. Roy", "MR. Das", "MR Doglus" ], "community_members" : [ 50
0, 300, 1400 ], "friends" : { "valuled_friends_id" : [ "pal", "viki", "john" ],
"ban_friends_id" : [ "jalan", "monoj", "evan" ] } } }
{ "_id" : ObjectId("4e3d1f2af52b9bd78374f954"), "user_id" : "user3", "password"
: "b1c1d1", "sex" : "Female", "age" : 19, "date_of_join" : "16/10/2010", "educat
ion" : "M.C.A.", "profession" : "IT COR.", "interest" : "ART", "extra" : { "comm
unity_name" : [ "MODERN ART", "CLASSICAL ART", "WESTERN ART" ], "community_moder
_id" : [ "MR. Rifel", "MR. Sarma", "MR Bhatia" ], "community_members" : [ 5000,
2000, 1500 ], "friends" : { "valued_friends_id" : [ "philip", "anant", "alan" ],
 "ban_friends_id" : [ "Amir", "Raja", "mont" ] } } }
{ "_id" : ObjectId("4e3d1f35f52b9bd78374f955"), "user_id" : "user4", "password"
: "abczyx", "sex" : "Female", "age" : 22, "date_of_join" : "17/8/2009", "educati
on" : "M.B.B.S.", "profession" : "DOCTOR", "interest" : "SPORTS", "extra" : { "c
ommunity_name" : [ "ATHELATIC", "GAMES FAN GYES", "FAVOURIT GAMES" ], "community
_moder_id" : [ "MR. Paul", "MR. Das", "MR Doglus" ], "community_members" : [ 250
0, 2200, 3500 ], "friends" : { "valued_friends_id" : [ "vinod", "viki", "john" ]
, "ban_friends_id" : [ "jalan", "monoj", "evan" ] } } }
>
>
```

更多实例

查询所有name字段是字符类型的数据：

```
db.users.find({name: {$type: 2}});
```

查询所有age字段是整型的数据：

```
db.users.find({age: {$type: 16}});
```

MongoDB Limit与Skip方法

MongoDB Limit() 方法

如果你需要在MongoDB中读取指定数量的数据记录，可以使用MongoDB的Limit方法，limit()方法接受一个数字参数，该参数指定从MongoDB中读取的记录条数。

语法

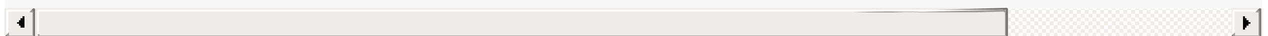
limit()方法基本语法如下所示：

```
>db.COLLECTION_NAME.find().limit(NUMBER)
```

实例

集合 myycol 中的数据如下：

```
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overvie
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview'
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point
```



以上实例为显示查询文档中的两条记录：

```
>db.mycol.find({},{"title":1,_id:0}).limit(2)
{"title":"MongoDB Overview"}
{"title":"NoSQL Overview"}
>
```

注：如果你们有指定limit()方法中的参数则显示集合中的所有数据。

MongoDB Skip() 方法

我们除了可以使用limit()方法来读取指定数量的数据外，还可以使用skip()方法来跳过指定数量的数据，skip方法同样接受一个数字参数作为跳过的记录条数。

语法

skip() 方法脚本语法格式如下：

```
>db.COLLECTION_NAME.find().limit(NUMBER).skip(NUMBER)
```

实例

以上实例只会显示第二条文档数据

```
>db.mycol.find({},{"title":1,_id:0}).limit(1).skip(1)
{"title":"NoSQL Overview"}
>
```

注:skip()方法默认参数为 0 。

MongoDB 排序

MongoDB sort()方法

在MongoDB中使用使用sort()方法对数据进行排序，sort()方法可以通过参数指定排序的字段，并使用 1 和 -1 来指定排序的方式，其中 1 为升序排序，而-1是用于降序排列。

语法

sort()方法基本语法如下所示：

```
>db.COLLECTION_NAME.find().sort({KEY:1})
```

实例

myycol 集合中的数据如下：

```
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overvie
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point
```

以下实例演示了 myycol 集合中的数据按字段 title 的降序排序：

```
>db.mycol.find({},{"title":1,_id:0}).sort({"title":-1})
{"title":"Tutorials Point Overview"}
{"title":"NoSQL Overview"}
{"title":"MongoDB Overview"}
>
```

注： 如果没有指定sort()方法的排序方式，默认按照文档的升序排序。

MongoDB 索引

索引通常能够极大的提高查询的效率，如果没有索引，MongoDB在读取数据时必须扫描集合中的每个文件并选取那些符合查询条件的记录。

这种扫描全集合的查询效率是非常低的，特别在处理大量的数据时，查询可以要花费几十秒甚至几分钟，这对网站的性能是非常致命的。

索引是特殊的数据结构，索引存储在一个易于遍历读取的数据集合中，索引是对数据库表中一列或多列的值进行排序的一种结构

ensureIndex() 方法

MongoDB使用 ensureIndex() 方法来创建索引。

语法

ensureIndex()方法基本语法格式如下所示：

```
>db.COLLECTION_NAME.ensureIndex({KEY:1})
```

语法中 Key 值为你要创建的索引字段，1为指定按升序创建索引，如果你想按降序来创建索引指定为-1即可。

实例

```
>db.mycol.ensureIndex({"title":1})
>
```

ensureIndex() 方法中你也可以设置使用多个字段创建索引（关系型数据库中称作复合索引）。

```
>db.mycol.ensureIndex({"title":1,"description":-1})
>
```

ensureIndex() 接收可选参数，可选参数列表如下：

Parameter	Type	Description
background	Boolean	建索引过程会阻塞其它数据库操作，background可指定以后台方式创建索引，即增加 "background" 可选参数。 "background" 默认值为**false**。
unique	Boolean	建立的索引是否唯一。指定为true创建唯一索引。默认值为**false**.
name	string	索引的名称。如果未指定，MongoDB的通过连接索引的字段名和排序顺序生成一个索引名称。
dropDups	Boolean	在建立唯一索引时是否删除重复记录,指定 true 创建唯一索引。默认值为 **false**.
sparse	Boolean	对文档中不存在的字段数据不启用索引；这个参数需要特别注意，如果设置为true的话，在索引字段中不会查询出不包含对应字段的文档.。默认值为 **false**.
expireAfterSeconds	integer	指定一个以秒为单位的数值，完成 TTL设定，设定集合的生存时间。
v	index version	索引的版本号。默认的索引版本取决于mongod创建索引时运行的版本。
weights	document	索引权重值，数值在 1 到 99,999 之间，表示该索引相对于其他索引字段的得分权重。
default_language	string	对于文本索引，该参数决定了停用词及词干和词器的规则的列表。 默认为英语
language_override	string	对于文本索引，该参数指定了包含在文档中的字段名，语言覆盖默认的language，默认值为 language.

实例

在后台创建索引：

```
db.values.ensureIndex({open: 1, close: 1}, {background: true})
```

通过在创建索引时加background:true 的选项，让创建工作在后台执行

MongoDB 聚合

MongoDB中聚合(aggregate)主要用于处理数据(诸如统计平均值,求和等)，并返回计算后的数据结果。有点类似sql语句中的 count(*)。

aggregate() 方法

MongoDB中聚合的方法使用aggregate()。

语法

aggregate() 方法的基本语法格式如下所示：

```
>db.COLLECTION_NAME.aggregate(AGGREGATE_OPERATION)
```

实例

集合中的数据如下：

```
{
   _id: ObjectId(7df78ad8902c)
   title: 'MongoDB Overview',
   description: 'MongoDB is no sql database',
   by_user: 'w3cschool.cc',
   url: 'http://www.w3cschool.cc',
   tags: ['mongodb', 'database', 'NoSQL'],
   likes: 100
},
{
   _id: ObjectId(7df78ad8902d)
   title: 'NoSQL Overview',
   description: 'No sql database is very fast',
   by_user: 'w3cschool.cc',
   url: 'http://www.w3cschool.cc',
   tags: ['mongodb', 'database', 'NoSQL'],
   likes: 10
},
{
   _id: ObjectId(7df78ad8902e)
   title: 'Neo4j Overview',
   description: 'Neo4j is no sql database',
   by_user: 'Neo4j',
   url: 'http://www.neo4j.com',
   tags: ['neo4j', 'database', 'NoSQL'],
   likes: 750
},
```

现在我们通过以上集合计算每个作者所写的文章数，使用aggregate()计算结果如下：

```
> db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : 
{
   "result" : [
      {
         "_id" : "w3cschool.cc",
         "num_tutorial" : 2
      },
      {
         "_id" : "Neo4j",
         "num_tutorial" : 1
      }
   ],
   "ok" : 1
}
>
```

以上实例类似sql语句： *select by_user, count(*) from mycol group by by_user*

在上面的例子中，我们通过字段by_user字段对数据进行分组，并计算by_user字段相同值的总和。

下表展示了一些聚合的表达式:

表达式	描述	实例
$sum	计算总和。	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$sum : "$likes"}}}])
$avg	计算平均值	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$avg : "$likes"}}}])
$min	获取集合中所有文档对应值得最小值。	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$min : "$likes"}}}])
$max	获取集合中所有文档对应值得最大值。	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$max : "$likes"}}}])
$push	在结果文档中插入值到一个数组中。	db.mycol.aggregate([{$group : {_id : "$by_user", url : {$push: "$url"}}}])
$addToSet	在结果文档中插入值到一个数组中，但不创建副本。	db.mycol.aggregate([{$group : {_id : "$by_user", url : {$addToSet : "$url"}}}])
$first	根据资源文档的排序获取第一个文档数据。	db.mycol.aggregate([{$group : {_id : "$by_user", first_url : {$first : "$url"}}}])
$last	根据资源文档的排序获取最后一个文档数据	db.mycol.aggregate([{$group : {_id : "$by_user", last_url : {$last : "$url"}}}])

管道的概念

管道在Unix和Linux中一般用于将当前命令的输出结果作为下一个命令的参数。

MongoDB的聚合管道将MongoDB文档在一个管道处理完毕后将结果传递给下一个管道处理。管道操作是可以重复的。

表达式：处理输入文档并输出。表达式是无状态的，只能用于计算当前聚合管道的文档，不能处理其它的文档。

这里我们介绍一下聚合框架中常用的几个操作：

- $project：修改输入文档的结构。可以用来重命名、增加或删除域，也可以用于创建计算结果以及嵌套文档。

- $match：用于过滤数据，只输出符合条件的文档。$match使用MongoDB的标准查询操作。
- $limit：用来限制MongoDB聚合管道返回的文档数。
- $skip：在聚合管道中跳过指定数量的文档，并返回余下的文档。
- $unwind：将文档中的某一个数组类型字段拆分成多条，每条包含数组中的一个值。
- $group：将集合中的文档分组，可用于统计结果。
- $sort：将输入文档排序后输出。
- $geoNear：输出接近某一地理位置的有序文档。

管道操作符实例

1、$project实例

```
db.article.aggregate(
    { $project : {
        title : 1 ,
        author : 1 ,
    }}
 );
```

这样的话结果中就只还有_id,tilte和author三个字段了，默认情况下_id字段是被包含的，如果要想不包含_id话可以这样:

```
db.article.aggregate(
    { $project : {
        _id : 0 ,
        title : 1 ,
        author : 1
    }});
```

2.$match实例

```
db.articles.aggregate( [
                        { $match : { score : { $gt : 70, $lte : 90
                        { $group: { _id: null, count: { $sum: 1 } ]
                       ] );
```

$match用于获取分数大于70小于或等于90记录，然后将符合条件的记录送到下一阶段$group管道操作符进行处理。

3.$skip实例

```
db.article.aggregate(
    { $skip : 5 });
```

经过$skip管道操作符处理后，前五个文档被"过滤"掉。

MongoDB 复制（副本集）

MongoDB复制是将数据同步在多个服务器的过程。

复制提供了数据的冗余备份，并在多个服务器上存储数据副本，提高了数据的可用性， 并可以保证数据的安全性。

复制还允许您从硬件故障和服务中断中恢复数据。

什么是复制?

- 保障数据的安全性
- 数据高可用性 (24*7)
- 灾难恢复
- 无需停机维护（如备份，重建索引，压缩）
- 分布式读取数据

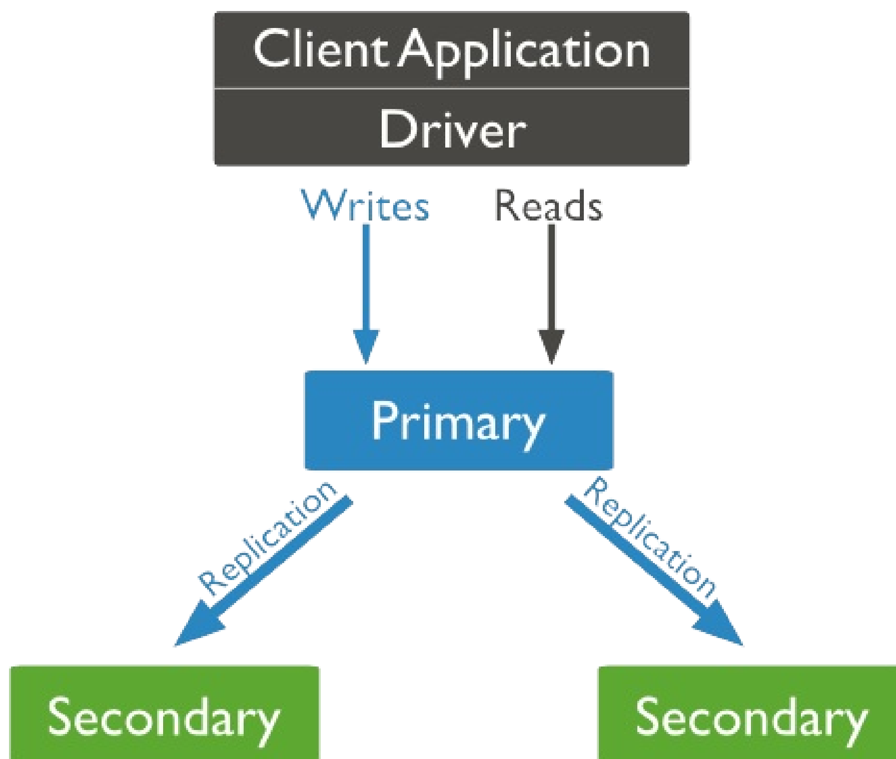
MongoDB复制原理

mongodb的复制至少需要两个节点。其中一个是主节点，负责处理客户端请求，其余的都是从节点，负责复制主节点上的数据。

mongodb各个节点常见的搭配方式为：一主一从、一主多从。

主节点记录在其上的所有操作oplog，从节点定期轮询主节点获取这些操作，然后对自己的数据副本执行这些操作，从而保证从节点的数据与主节点一致。

MongoDB复制结构图如下所示：



以上结构图总，客户端总主节点读取数据，在客户端写入数据到主节点是， 主节点与从节点进行数据交互保障数据的一致性。

副本集特征：

- N 个节点的集群
- 任何节点可作为主节点
- 所有写入操作都在主节点上
- 自动故障转移
- 自动恢复

MongoDB副本集设置

在本教程中我们使用同一个MongoDB来做MongoDB主从的实验， 操作步骤如下：

1、关闭正在运行的MongoDB服务器。

现在我们通过指定 --replSet 选项来启动mongoDB。--replSet 基本语法格式如下：

```
mongod --port "PORT" --dbpath "YOUR_DB_DATA_PATH" --replSet "REPLIC
```

实例

```
mongod --port 27017 --dbpath "D:\set up\mongodb\data" --replSet rs0
```

以上实例会启动一个名为rs0的MongoDB实例，其端口号为27017。

启动后打开命令提示框并连接上mongoDB服务。

在Mongo客户端使用命令rs.initiate()来启动一个新的副本集。

我们可以使用rs.conf()来查看副本集的配置

查看副本集姿态使用 rs.status() 命令

副本集添加成员

添加副本集的成员，我们需要使用多条服务器来启动mongo服务。进入Mongo客户端，并使用rs.add()方法来添加副本集的成员。

语法

rs.add() 命令基本语法格式如下：

```
>rs.add(HOST_NAME:PORT)
```

实例

假设你已经启动了一个名为mongod1.net，端口号为27017的Mongo服务。 在客户端命令窗口使用rs.add() 命令将其添加到副本集中，命令如下所示：

```
>rs.add("mongod1.net:27017")
>
```

MongoDB中你只能通过主节点将Mongo服务添加到副本集中， 判断当前运行的Mongo服务是否为主节点可以使用命令db.isMaster() 。

MongoDB的副本集与我们常见的主从有所不同，主从在主机宕机后所有服务将停止，而副本集在主机宕机后，副本会接管主节点成为主节点，不会出现宕机的情况。

MongoDB 分片

分片

在Mongodb里面存在另一种集群，就是分片技术,可以满足MongoDB数据量大量增长的需求。

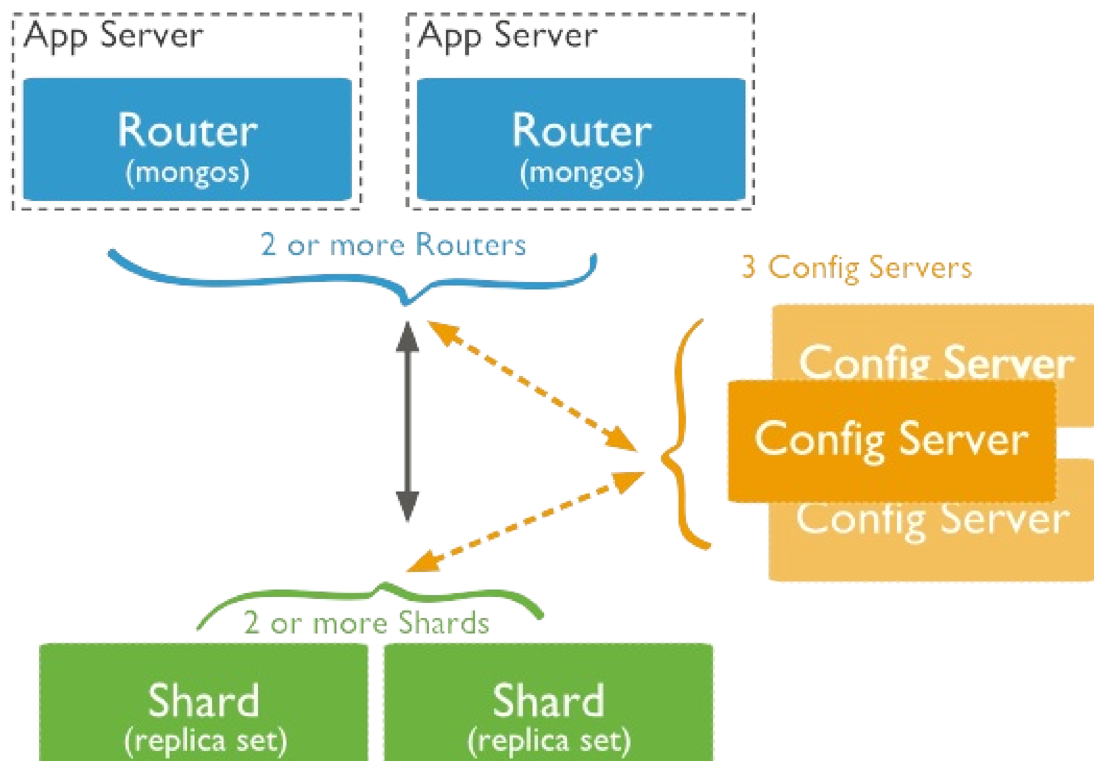
当MongoDB存储海量的数据时，一台机器可能不足以存储数据也足以提供可接受的读写吞吐量。这时，我们就可以通过在多台机器上分割数据，使得数据库系统能存储和处理更多的数据。

为什么使用分片

- 复制所有的写入操作到主节点
- 延迟的敏感数据会在主节点查询
- 单个副本集限制在12个节点
- 当请求量巨大时会出现内存不足。
- 本地磁盘不足
- 垂直扩展价格昂贵

MongoDB分片

下图展示了在MongoDB中使用分片集群结构分布：



上图中主要有如下所述三个主要组件：

- **Shard:**

  用于存储实际的数据块，实际生产环境中一个shard server角色可由几台机器组个一个relica set承担，防止主机单点故障

- **Config Server:**

  mongod实例，存储了整个 ClusterMetadata，其中包括 chunk信息。

- **Query Routers:**

  前端路由，客户端由此接入，且让整个集群看上去像单一数据库，前端应用可以透明使用。

分片实例

分片结构端口分布如下：

```
Shard Server 1：27020
Shard Server 2：27021
Shard Server 3：27022
Shard Server 4：27023
Config Server ：27100
Route Process：40000
```

步骤一：启动Shard Server

```
[root@100 /]# mkdir -p /www/mongoDB/shard/s0
[root@100 /]# mkdir -p /www/mongoDB/shard/s1
[root@100 /]# mkdir -p /www/mongoDB/shard/s2
[root@100 /]# mkdir -p /www/mongoDB/shard/s3
[root@100 /]# mkdir -p /www/mongoDB/shard/log
[root@100 /]# /usr/local/mongoDB/bin/mongod --port 27020 --dbpath=/
....
[root@100 /]# /usr/local/mongoDB/bin/mongod --port 27023 --dbpath=/
```

步骤二： 启动Config Server

```
[root@100 /]# mkdir -p /www/mongoDB/shard/config
[root@100 /]# /usr/local/mongoDB/bin/mongod --port 27100 --dbpath=/
```

注意：这里我们完全可以像启动普通mongodb服务一样启动，不需要添加—shardsvr和configsvr参数。因为这两个参数的作用就是改变启动端口的，所以我们自行指定了端口就可以。

步骤三： 启动Route Process

```
/usr/local/mongoDB/bin/mongos --port 40000 --configdb localhost:271
```

mongos启动参数中，chunkSize这一项是用来指定chunk的大小的，单位是MB，默认大小为200MB.

步骤四： 配置Sharding

接下来，我们使用MongoDB Shell登录到mongos，添加Shard节点

```
[root@100 shard]# /usr/local/mongoDB/bin/mongo admin --port 40000
MongoDB shell version: 2.0.7
connecting to: 127.0.0.1:40000/admin
mongos> db.runCommand({ addshard:"localhost:27020" })
{ "shardAdded" : "shard0000", "ok" : 1 }
......
mongos> db.runCommand({ addshard:"localhost:27029" })
{ "shardAdded" : "shard0009", "ok" : 1 }
mongos> db.runCommand({ enablesharding:"test" }) #设置分片存储的数据库
{ "ok" : 1 }
mongos> db.runCommand({ shardcollection: "test.log", key: { id:1,ti
{ "collectionsharded" : "test.log", "ok" : 1 }
```

步骤五： 程序代码内无需太大更改，直接按照连接普通的mongo数据库那样，将数据库连接接入接口40000

MongoDB 备份(mongodump)与恢复(mongorerstore)

MongoDB数据备份

在Mongodb中我们使用mongodump命令来备份MongoDB数据。该命令可以导出所有数据到指定目录中。

mongodump命令可以通过参数指定导出的数据量级转存的服务器。

语法

mongodump命令脚本语法如下：

```
>mongodump -h dbhost -d dbname -o dbdirectory
```

- **-h**：

  MongDB所在服务器地址，例如：127.0.0.1，当然也可以指定端口号：127.0.0.1:27017

- **-d**：

  需要备份的数据库实例，例如：test

- **-o**：

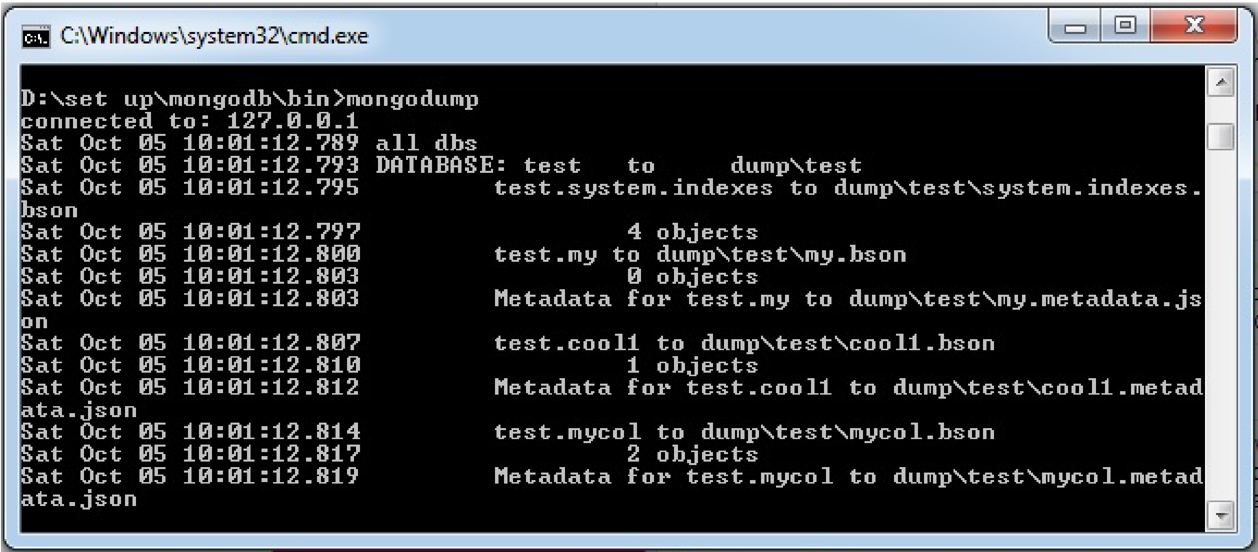
  备份的数据存放位置，例如：c:\data\dump，当然该目录需要提前建立，在备份完成后，系统自动在dump目录下建立一个test目录，这个目录里面存放该数据库实例的备份数据。

实例

在本地使用 27017 启动你的mongod服务。打开命令提示符窗口，进入MongoDB安装目录的bin目录输入命令mongodump:

```
>mongodump
```

执行以上命令后，客户端会连接到ip为 127.0.0.1 端口号为 27017 的MongoDB服务上，并备份所有数据到 bin/dump/ 目录中。命令输出结果如下：



mongodump 命令可选参数列表如下所示：

语法	描述	实例
mongodump --host HOST_NAME --port PORT_NUMBER	该命令将备份所有MongoDB数据	mongodump --host w3cschool.cc --port 27017
mongodump --dbpath DB_PATH --out BACKUP_DIRECTORY		mongodump --dbpath /data/db/ --out /data/backup/
mongodump --collection COLLECTION --db DB_NAME	该命令将备份指定数据库的集合。	mongodump --collection mycol --db test

MongoDB数据恢复

mongodb使用 mongorerstore 命令来恢复备份的数据。

语法

mongorestore命令脚本语法如下：

```
>mongorestore -h dbhost -d dbname --directoryperdb dbdirectory
```

- **-h**：

  MongoDB所在服务器地址

- **-d**：

需要恢复的数据库实例，例如：test，当然这个名称也可以和备份时候的不一样，比如test2

- **--directoryperdb**：

  备份数据所在位置，例如：c:\data\dump\test，这里为什么要多加一个test，而不是备份时候的dump，读者自己查看提示吧！

- **--drop**：

  恢复的时候，先删除当前数据，然后恢复备份的数据。就是说，恢复后，备份后添加修改的数据都会被删除，慎用哦！

接下来我们执行以下命令:

```
>mongorestore
```

执行以上命令输出结果如下：

```
C:\Windows\system32\cmd.exe
D:\set up\mongodb\bin>mongorestore
connected to: 127.0.0.1
Sat Oct 05 10:06:40.922 dump\test\cool1.bson
Sat Oct 05 10:06:40.924         going into namespace [test.cool1]
Sat Oct 05 10:06:40.933 warning: Restoring to test.cool1 without dropping. Resto
red data will be inserted without raising errors; check your server log
1 objects found
Sat Oct 05 10:06:41.003         Creating index: { key: { _id: 1 }, ns: "test.coo
l1", name: "_id_" }
Sat Oct 05 10:06:41.058 dump\test\my.bson
Sat Oct 05 10:06:41.058         going into namespace [test.my]
Sat Oct 05 10:06:41.062 warning: Restoring to test.my without dropping. Restored
 data will be inserted without raising errors; check your server log
Sat Oct 05 10:06:41.063 file dump\test\my.bson empty, skipping
Sat Oct 05 10:06:41.063         Creating index: { key: { _id: 1 }, ns: "test.my"
, name: "_id_" }
Sat Oct 05 10:06:41.066 dump\test\mycol.bson
Sat Oct 05 10:06:41.067         going into namespace [test.mycol]
Sat Oct 05 10:06:41.070 warning: Restoring to test.mycol without dropping. Resto
red data will be inserted without raising errors; check your server log
2 objects found
Sat Oct 05 10:06:41.077         Creating index: { key: { _id: 1 }, ns: "test.myc
ol", name: "_id_" }
Sat Oct 05 10:06:41.079         Creating index: { key: { name: 1 }, ns: "test.my
col", name: "name_1" }
```

MongoDB 监控

在你已经安装部署并允许MongoDB服务后，你必须要了解MongoDB的运行情况，并查看MongoDB的性能。这样在大流量得情况下可以很好的应对并保证MongoDB正常运作。

MongoDB中提供了mongostat 和 mongotop 两个命令来监控MongoDB的运行情况。

mongostat 命令

mongostat是mongodb自带的状态检测工具，在命令行下使用。它会间隔固定时间获取mongodb的当前运行状态，并输出。如果你发现数据库突然变慢或者有其他问题的话，你第一手的操作就考虑采用mongostat来查看mongo的状态。

启动你的Mongod服务，进入到你安装的MongoDB目录下的bin目录， 然后输入mongostat命令，如下所示：

```
D:\set up\mongodb\bin>mongostat
```

以上命令输出结果如下：

```
local:0.0%          0       0|0     0|0    115b     4k      2   19:20:24
insert  query update delete getmore command flushes mapped  vsize    res faults
  locked db idx miss %     qr|qw   ar|aw  netIn netOut  conn       time
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0    115b     4k      2   19:20:25
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0    115b     4k      2   19:20:26
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0    115b     4k      2   19:20:27
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0    115b     4k      2   19:20:28
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0    115b     4k      2   19:20:29
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0    115b     4k      2   19:20:30
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0    115b     4k      2   19:20:31
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0    115b     4k      2   19:20:32
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0    115b     4k      2   19:20:33
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0    115b     4k      2   19:20:34
insert  query update delete getmore command flushes mapped  vsize    res faults
  locked db idx miss %     qr|qw   ar|aw  netIn netOut  conn       time
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0    115b     4k      2   19:20:35
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0    115b     4k      2   19:20:36
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0    115b     4k      2   19:20:37
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0    115b     4k      2   19:20:38
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0    115b     4k      2   19:20:39
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0    115b     4k      2   19:20:40
    *0     *0     *0     *0       0     2|0       0  14.1g  28.3g    40m      0
local:0.0%          0       0|0     0|0    115b     4k      2   19:20:41
```

mongotop 命令

mongotop也是mongodb下的一个内置工具，mongotop提供了一个方法，用来跟踪一个MongoDB的实例，查看哪些大量的时间花费在读取和写入数据。 mongotop提供每个集合的水平的统计数据。默认情况下，mongotop返回值的每一秒。

启动你的Mongod服务，进入到你安装的MongoDB目录下的bin目录， 然后输入mongotop命令，如下所示：

```
D:\set up\mongodb\bin>mongotop
```

以上命令执行输出结果如下：

```
C:\Windows\system32\cmd.exe - mongotop
            local.system.users         0ms         0ms         0ms
           local.system.replset        0ms         0ms         0ms
             local.startup_log         0ms         0ms         0ms

                            ns       total        read       write
2013-10-06T13:53:28
             test.system.users         0ms         0ms         0ms
            local.system.users         0ms         0ms         0ms
          local.system.replset         0ms         0ms         0ms
             local.startup_log         0ms         0ms         0ms

                            ns       total        read       write
2013-10-06T13:53:29
             test.system.users         0ms         0ms         0ms
            local.system.users         0ms         0ms         0ms
          local.system.replset         0ms         0ms         0ms
             local.startup_log         0ms         0ms         0ms

                            ns       total        read       write
2013-10-06T13:53:30
             test.system.users         0ms         0ms         0ms
            local.system.users         0ms         0ms         0ms
          local.system.replset         0ms         0ms         0ms
             local.startup_log         0ms         0ms         0ms
```

带参数实例

```
E:\mongodb-win32-x86_64-2.2.1\bin>mongotop 10
```

```
E:\mongodb-win32-x86_64-2.2.1\bin>mongotop
connected to: 127.0.0.1

                                  ns       total        read       write
2013-03-29T04:12:17
                local.system.replset         0ms         0ms         0ms
             local.system.namespaces         0ms         0ms         0ms
                local.system.indexes         0ms         0ms         0ms
                admin.system.indexes         0ms         0ms         0ms
                              admin.         0ms         0ms         0ms
CpsCommodityInfo.system.namespaces         0ms         0ms         0ms
        CpsCommodityInfo.system.js         0ms         0ms         0ms

                                  ns       total        read       write
2013-03-29T04:12:18
                local.system.replset         0ms         0ms         0ms
             local.system.namespaces         0ms         0ms         0ms
                local.system.indexes         0ms         0ms         0ms
                admin.system.indexes         0ms         0ms         0ms
                              admin.         0ms         0ms         0ms
CpsCommodityInfo.system.namespaces         0ms         0ms         0ms
        CpsCommodityInfo.system.js         0ms         0ms         0ms
```

后面的10是*<sleeptime>*参数，可以不使用，等待的时间长度，以秒为单位，mongotop等待调用之间。通过的默认mongotop返回数据的每一秒。

```
E:\mongodb-win32-x86_64-2.2.1\bin>mongotop --locks
```

报告每个数据库的锁的使用中，使用mongotop - 锁，这将产生以下输出：

```
E:\mongodb-win32-x86_64-2.2.1\bin>mongotop --locks
connected to: 127.0.0.1

                          db       total        read       write
2013-03-29T04:21:59
                       local         0ms         0ms         0ms
                       admin         0ms         0ms         0ms
            CpsCommodityInfo         0ms         0ms         0ms
                           .         0ms         0ms         0ms

                          db       total        read       write
2013-03-29T04:22:00
                       local         0ms         0ms         0ms
                       admin         0ms         0ms         0ms
            CpsCommodityInfo         0ms         0ms         0ms
                           .         0ms         0ms         0ms

                          db       total        read       write
2013-03-29T04:22:01
```

输出结果字段说明：

- **ns**：

  包含数据库命名空间，后者结合了数据库名称和集合。

- **db**：

  包含数据库的名称。名为 . 的数据库针对全局锁定，而非特定数据库。

- **total**：

  mongod花费的时间工作在这个命名空间提供总额。

- **read**：

  提供了大量的时间，这mongod花费在执行读操作，在此命名空间。

- **write**：

  提供这个命名空间进行写操作，这mongod花了大量的时间。

MongoDB Java

环境配置

在Java程序中如果要使用MongoDB，你需要确保已经安装了Java环境及MongoDB JDBC 驱动。

你可以参考本站的Java教程来安装Java程序。现在让我们来检测你是否安装了MongoDB JDBC 驱动。

- 首先你必须下载mongo jar包，下载地址：https://github.com/mongodb/mongo-java-driver/downloads, 请确保下载最新版本。
- 你需要将mongo.jar包含在你的 classpath 中。。

连接数据库

连接数据库，你需要指定数据库名称，如果指定的数据库不存在，mongo会自动创建数据库。

连接数据库的Java代码如下：

```
import com.mongodb.MongoClient;
import com.mongodb.MongoException;
import com.mongodb.WriteConcern;
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.BasicDBObject;
import com.mongodb.DBObject;
import com.mongodb.DBCursor;
import com.mongodb.ServerAddress;
import java.util.Arrays;

public class MongoDBJDBC{
   public static void main( String args[] ){
      try{
         // 连接到 mongodb 服务
         MongoClient mongoClient = new MongoClient( "localhost" , 2
         // 连接到数据库
         DB db = mongoClient.getDB( "test" );
         System.out.println("Connect to database successfully");
         boolean auth = db.authenticate(myUserName, myPassword);
         System.out.println("Authentication: "+auth);
      }catch(Exception e){
         System.err.println( e.getClass().getName() + ": " + e.getM
      }
   }
}
```

现在，让我们来编译运行程序并创建数据库test。

你可以更加你的实际环境改变MongoDB JDBC驱动的路径。

本实例将MongoDB JDBC启动包 mongo-2.10.1.jar 放在本地目录下:

```
$javac MongoDBJDBC.java
$java -classpath ".:mongo-2.10.1.jar" MongoDBJDBC
Connect to database successfully
Authentication: true
```

如果你使用的是Window系统，你可以按以下命令来编译执行程序：

```
$javac MongoDBJDBC.java
$java -classpath ".;mongo-2.10.1.jar" MongoDBJDBC
Connect to database successfully
Authentication: true
```

如果用户名及密码正确，则Authentication 的值为true。

创建集合

我们可以使用com.mongodb.DB类中的createCollection()来创建集合

代码片段如下：

```
import com.mongodb.MongoClient;
import com.mongodb.MongoException;
import com.mongodb.WriteConcern;
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.BasicDBObject;
import com.mongodb.DBObject;
import com.mongodb.DBCursor;
import com.mongodb.ServerAddress;
import java.util.Arrays;

public class MongoDBJDBC{
   public static void main( String args[] ){
      try{
         // 连接到 mongodb 服务
         MongoClient mongoClient = new MongoClient( "localhost" , 2
         // 连接到数据库
         DB db = mongoClient.getDB( "test" );
     System.out.println("Connect to database successfully");
         boolean auth = db.authenticate(myUserName, myPassword);
     System.out.println("Authentication: "+auth);
         DBCollection coll = db.createCollection("mycol");
         System.out.println("Collection created successfully");
      }catch(Exception e){
         System.err.println( e.getClass().getName() + ": " + e.getM
      }
   }
}
```

编译运行以上程序，输出结果如下:

```
Connect to database successfully
Authentication: true
Collection created successfully
```

获取集合

我们可以使用com.mongodb.DBCollection类的 getCollection() 方法来获取一个集合

代码片段如下：

```
import com.mongodb.MongoClient;
import com.mongodb.MongoException;
import com.mongodb.WriteConcern;
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.BasicDBObject;
import com.mongodb.DBObject;
import com.mongodb.DBCursor;
import com.mongodb.ServerAddress;
import java.util.Arrays;

public class MongoDBJDBC{
   public static void main( String args[] ){
      try{
         // 连接到 mongodb 服务
         MongoClient mongoClient = new MongoClient( "localhost" , 2
         // 连接到数据库
         DB db = mongoClient.getDB( "test" );
     System.out.println("Connect to database successfully");
         boolean auth = db.authenticate(myUserName, myPassword);
     System.out.println("Authentication: "+auth);
         DBCollection coll = db.createCollection("mycol");
         System.out.println("Collection created successfully");
         DBCollection coll = db.getCollection("mycol");
         System.out.println("Collection mycol selected successfully
      }catch(Exception e){
         System.err.println( e.getClass().getName() + ": " + e.getM
      }
   }
}
```

编译运行以上程序，输出结果如下:

```
Connect to database successfully
Authentication: true
Collection created successfully
Collection mycol selected successfully
```

插入文档

我们可以使用com.mongodb.DBCollection类的 insert() 方法来插入一个文档

代码片段如下：

```
import com.mongodb.MongoClient;
import com.mongodb.MongoException;
import com.mongodb.WriteConcern;
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.BasicDBObject;
import com.mongodb.DBObject;
import com.mongodb.DBCursor;
import com.mongodb.ServerAddress;
import java.util.Arrays;

public class MongoDBJDBC{
   public static void main( String args[] ){
      try{
         // 连接到 mongodb 服务
         MongoClient mongoClient = new MongoClient( "localhost" , 2
         // 连接到数据库
         DB db = mongoClient.getDB( "test" );
     System.out.println("Connect to database successfully");
         boolean auth = db.authenticate(myUserName, myPassword);
     System.out.println("Authentication: "+auth);
         DBCollection coll = db.getCollection("mycol");
         System.out.println("Collection mycol selected successfully
         BasicDBObject doc = new BasicDBObject("title", "MongoDB").
            append("description", "database").
            append("likes", 100).
            append("url", "http://www.w3cschool.cc/mongodb/").
            append("by", "w3cschool.cc");
         coll.insert(doc);
         System.out.println("Document inserted successfully");
      }catch(Exception e){
         System.err.println( e.getClass().getName() + ": " + e.getM
      }
   }
}
```

编译运行以上程序，输出结果如下:

```
Connect to database successfully
Authentication: true
Collection mycol selected successfully
Document inserted successfully
```

检索所有文档

我们可以使用com.mongodb.DBCollection类中的 find() 方法来获取集合中的所有文档。

此方法返回一个游标，所以你需要遍历这个游标。

代码片段如下：

```
import com.mongodb.MongoClient;
import com.mongodb.MongoException;
import com.mongodb.WriteConcern;
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.BasicDBObject;
import com.mongodb.DBObject;
import com.mongodb.DBCursor;
import com.mongodb.ServerAddress;
import java.util.Arrays;

public class MongoDBJDBC{
   public static void main( String args[] ){
      try{
        // 连接到 mongodb 服务
         MongoClient mongoClient = new MongoClient( "localhost" , 2
          // 连接到数据库
         DB db = mongoClient.getDB( "test" );
     System.out.println("Connect to database successfully");
         boolean auth = db.authenticate(myUserName, myPassword);
     System.out.println("Authentication: "+auth);
         DBCollection coll = db.getCollection("mycol");
         System.out.println("Collection mycol selected successfully
         DBCursor cursor = coll.find();
         int i=1;
         while (cursor.hasNext()) {
            System.out.println("Inserted Document: "+i);
            System.out.println(cursor.next());
            i++;
         }
      }catch(Exception e){
         System.err.println( e.getClass().getName() + ": " + e.getM
      }
   }
}
```

编译运行以上程序，输出结果如下:

```
Connect to database successfully
Authentication: true
Collection mycol selected successfully
Inserted Document: 1
{
   "_id" : ObjectId(7df78ad8902c),
   "title": "MongoDB",
   "description": "database",
   "likes": 100,
   "url": "http://www.w3cschool.cc/mongodb/",
   "by": "w3cschool.cc"
}
```

更新文档

你可以使用 com.mongodb.DBCollection 类中的 update() 方法来更新集合中的文档。

代码片段如下：

```
import com.mongodb.MongoClient;
import com.mongodb.MongoException;
import com.mongodb.WriteConcern;
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.BasicDBObject;
import com.mongodb.DBObject;
import com.mongodb.DBCursor;
import com.mongodb.ServerAddress;
import java.util.Arrays;

public class MongoDBJDBC{
   public static void main( String args[] ){
      try{
     // 连接到Mongodb服务
         MongoClient mongoClient = new MongoClient( "localhost" , 2
         // 连接到你的数据库
         DB db = mongoClient.getDB( "test" );
     System.out.println("Connect to database successfully");
         boolean auth = db.authenticate(myUserName, myPassword);
     System.out.println("Authentication: "+auth);
         DBCollection coll = db.getCollection("mycol");
         System.out.println("Collection mycol selected successfully
         DBCursor cursor = coll.find();
         while (cursor.hasNext()) {
            DBObject updateDocument = cursor.next();
            updateDocument.put("likes","200")
            col1.update(updateDocument);
         }
         System.out.println("Document updated successfully");
         cursor = coll.find();
         int i=1;
         while (cursor.hasNext()) {
            System.out.println("Updated Document: "+i);
            System.out.println(cursor.next());
            i++;
         }
      }catch(Exception e){
         System.err.println( e.getClass().getName() + ": " + e.getM
      }
   }
}
```

编译运行以上程序，输出结果如下:

```
Connect to database successfully
Authentication: true
Collection mycol selected successfully
Document updated successfully
Updated Document: 1
{
   "_id" : ObjectId(7df78ad8902c),
   "title": "MongoDB",
   "description": "database",
   "likes": 100,
   "url": "http://www.w3cschool.cc/mongodb/",
   "by": "w3cschool.cc"
}
```

删除第一个文档

要删除集合中的第一个文档，首先你需要使用com.mongodb.DBCollection类中的findOne()方法来获取第一个文档，然后使用remove 方法删除。

代码片段如下：

```
 import com.mongodb.MongoClient;
import com.mongodb.MongoException;
import com.mongodb.WriteConcern;
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.BasicDBObject;
import com.mongodb.DBObject;
import com.mongodb.DBCursor;
import com.mongodb.ServerAddress;
import java.util.Arrays;

public class MongoDBJDBC{
   public static void main( String args[] ){
      try{
     // 连接到Mongodb服务
         MongoClient mongoClient = new MongoClient( "localhost" , 2
         // 连接到你的数据库
         DB db = mongoClient.getDB( "test" );
     System.out.println("Connect to database successfully");
         boolean auth = db.authenticate(myUserName, myPassword);
     System.out.println("Authentication: "+auth);
         DBCollection coll = db.getCollection("mycol");
         System.out.println("Collection mycol selected successfully
         DBObject myDoc = coll.findOne();
         col1.remove(myDoc);
         DBCursor cursor = coll.find();
         int i=1;
         while (cursor.hasNext()) {
            System.out.println("Inserted Document: "+i);
            System.out.println(cursor.next());
            i++;
         }
         System.out.println("Document deleted successfully");
      }catch(Exception e){
         System.err.println( e.getClass().getName() + ": " + e.getM
      }
   }
}
```

编译运行以上程序，输出结果如下:

```
Connect to database successfully
Authentication: true
Collection mycol selected successfully
Document deleted successfully
```

你还可以使用 save(), limit(), skip(), sort() 等方法来操作MongoDB数据库。

MongoDB PHP

在php中使用mongodb你必须使用 mongodb的php驱动。

MongoDB PHP在各平台上的安装及驱动包下载请查看:PHP安装MongoDB扩展驱动

确保连接及选择一个数据库

为了确保正确连接，你需要指定数据库名，如果数据库在mongoDB中不存在，mongoDB会自动创建

代码片段如下：

```
<?php
   // 连接到mongodb
   $m = new MongoClient();
   echo "Connection to database successfully";
   // 选择一个数据库
   $db = $m->mydb;
   echo "Database mydb selected";
?>
```

执行以上程序，输出结果如下：

```
Connection to database successfully
Database mydb selected
```

创建集合

创建集合的代码片段如下：

```
<?php
   // 连接到mongodb
   $m = new MongoClient();
   echo "Connection to database successfully";
   // 选择一个数据库
   $db = $m->mydb;
   echo "Database mydb selected";
   $collection = $db->createCollection("mycol");
   echo "Collection created succsessfully";
?>
```

执行以上程序，输出结果如下：

```
Connection to database successfully
Database mydb selected
Collection created succsessfully
```

插入文档

在mongoDB中使用 insert() 方法插入文档：

插入文档代码片段如下：

```
<?php
   // 连接到mongodb
   $m = new MongoClient();
   echo "Connection to database successfully";
   // 选择一个数据库
   $db = $m->mydb;
   echo "Database mydb selected";
   $collection = $db->mycol;
   echo "Collection selected succsessfully";
   $document = array(
      "title" => "MongoDB",
      "description" => "database",
      "likes" => 100,
      "url" => "http://www.w3cschool.cc/mongodb/",
      "by", "w3cschool.cc"
   );
   $collection->insert($document);
   echo "Document inserted successfully";
?>
```

执行以上程序，输出结果如下：

```
Connection to database successfully
Database mydb selected
Collection selected succsessfully
Document inserted successfully
```

查找文档

使用find() 方法来读取集合中的文档。

读取使用文档的代码片段如下：

```
<?php
   // 连接到mongodb
   $m = new MongoClient();
   echo "Connection to database successfully";
   // 选择一个数据库
   $db = $m->mydb;
   echo "Database mydb selected";
   $collection = $db->mycol;
   echo "Collection selected succsessfully";

   $cursor = $collection->find();
   // 迭代显示文档标题
   foreach ($cursor as $document) {
      echo $document["title"] . "\n";
   }
?>
```

执行以上程序，输出结果如下：

```
Connection to database successfully
Database mydb selected
Collection selected succsessfully
{
   "title": "MongoDB"
}
```

更新文档

使用 update() 方法来更新文档。

以下实例将更新文档中的标题为' MongoDB Tutorial'， 代码片段如下：

```
<pre>
<?php
   // 连接到mongodb
   $m = new MongoClient();
   echo "Connection to database successfully";
   // 选择一个数据库
   $db = $m->mydb;
   echo "Database mydb selected";
   $collection = $db->mycol;
   echo "Collection selected succsessfully";

   // 更新文档
   $collection->update(array("title"=>"MongoDB"), array('$set'=>ar
   echo "Document updated successfully";
   // 显示更新后的文档
   $cursor = $collection->find();
   // 循环显示文档标题
   echo "Updated document";
   foreach ($cursor as $document) {
      echo $document["title"] . "\n";
   }
?>
```

执行以上程序，输出结果如下：

```
Connection to database successfully
Database mydb selected
Collection selected succsessfully
Document updated successfully
Updated document
{
   "title": "MongoDB Tutorial"
}
```

删除文档

使用 remove() 方法来删除文档。

以下实例中我们将移除 'title' 为 'MongoDB Tutorial' 的数据记录。， 代码片段如下：

```
<?php
   // 连接到mongodb
   $m = new MongoClient();
   echo "Connection to database successfully";
   // 选择一个数据库
   $db = $m->mydb;
   echo "Database mydb selected";
   $collection = $db->mycol;
   echo "Collection selected succsessfully";

   // 移除文档
   $collection->remove(array("title"=>"MongoDB Tutorial"),false);
   echo "Documents deleted successfully";

   // 显示可用文档数据
   $cursor = $collection->find();
   // iterate cursor to display title of documents
   echo "Updated document";
   foreach ($cursor as $document) {
      echo $document["title"] . "\n";
   }
?>
```

执行以上程序，输出结果如下：

```
Connection to database successfully
Database mydb selected
Collection selected succsessfully
Documents deleted successfully
```

除了以上实例外，在php中你还可以使用findOne(), save(), limit(), skip(), sort()等方法来操作Mongodb数据库。

MongoDB 关系

MongoDB 的关系表示多个文档之间在逻辑上的相互联系。

文档间可以通过嵌入和引用来建立联系。

MongoDB 中的关系可以是：

- 1:1 (1对1)
- 1: N (1对多)
- N: 1 (多对1)
- N: N (多对多)

接下来我们来考虑下用户与用户地址的关系。

一个用户可以有多个地址，所以是一对多的关系。

以下是 **user** 文档的简单结构：

```
{
   "_id":ObjectId("52ffc33cd85242f436000001"),
   "name": "Tom Hanks",
   "contact": "987654321",
   "dob": "01-01-1991"
}
```

以下是 **address** 文档的简单结构：

```
{
   "_id":ObjectId("52ffc4a5d85242602e000000"),
   "building": "22 A, Indiana Apt",
   "pincode": 123456,
   "city": "Los Angeles",
   "state": "California"
}
```

嵌入式关系

使用嵌入式方法，我们可以把用户地址嵌入到用户的文档中：

```
    "_id":ObjectId("52ffc33cd85242f436000001"),
    "contact": "987654321",
    "dob": "01-01-1991",
    "name": "Tom Benzamin",
    "address": [
       {
          "building": "22 A, Indiana Apt",
          "pincode": 123456,
          "city": "Los Angeles",
          "state": "California"
       },
       {
          "building": "170 A, Acropolis Apt",
          "pincode": 456789,
          "city": "Chicago",
          "state": "Illinois"
       }]
}
```

以上数据保存在单一的文档中，可以比较容易的获取很维护数据。 你可以这样查询用户的地址：

```
>db.users.findOne({"name":"Tom Benzamin"},{"address":1})
```

注意：以上查询中 **db** 和 **users** 表示数据库和集合。

这种数据结构的缺点是，如果用户和用户地址在不断增加，数据量不断变大，会影响读写性能。

引用式关系

引用式关系是设计数据库时经常用到的方法，这种方法把用户数据文档和用户地址数据文档分开，通过引用文档的 **id** 字段来建立关系。

```
{
   "_id":ObjectId("52ffc33cd85242f436000001"),
   "contact": "987654321",
   "dob": "01-01-1991",
   "name": "Tom Benzamin",
   "address_ids": [
      ObjectId("52ffc4a5d85242602e000000"),
      ObjectId("52ffc4a5d85242602e000001")
   ]
}
```

以上实例中，用户文档的 **address_ids** 字段包含用户地址的对象id（ObjectId）数组。

我们可以读取这些用户地址的对象id（ObjectId）来获取用户的详细地址信息。

这种方法需要两次查询，第一次查询用户地址的对象id（ObjectId），第二次通过查询的id获取用户的详细地址信息。

```
>var result = db.users.findOne({"name":"Tom Benzamin"},{"address_ic
>var addresses = db.address.find({"_id":{"$in":result["address_ids'
```

MongoDB 数据库引用

在上一章节MongoDB关系中我们提到了MongoDB的引用来规范数据结构文档。

MongoDB 引用有两种：

- 手动引用（Manual References）
- DBRefs

DBRefs vs 手动引用

考虑这样的一个场景，我们在不同的集合中 (address_home, address_office, address_mailing, 等)存储不同的地址（住址，办公室地址，邮件地址等）。

这样，我们在调用不同地址时，也需要指定集合，一个文档从多个集合引用文档，我们应该使用 DBRefs。

使用 DBRefs

DBRef的形式：

```
{ $ref : , $id : , $db :  }
```

三个字段表示的意义为：

- $ref：集合名称
- $id：引用的id
- $db:数据库名称，可选参数

以下实例中用户数据文档使用了 DBRef, 字段 address：

```
{
   "_id":ObjectId("53402597d852426020000002"),
   "address": {
   "$ref": "address_home",
   "$id": ObjectId("534009e4d852427820000002"),
   "$db": "w3cschoolcc"},
   "contact": "987654321",
   "dob": "01-01-1991",
   "name": "Tom Benzamin"
}
```

address DBRef 字段指定了引用的地址文档是在 address_home 集合下的 w3cschoolcc 数据库，id 为 534009e4d852427820000002。

以下代码中，我们通过指定 $ref 参数（address_home 集合）来查找集合中指定id的用户地址信息：

```
>var user = db.users.findOne({"name":"Tom Benzamin"})
>var dbRef = user.address
>db[dbRef.$ref].findOne({"_id":(dbRef.$id)})
```

以上实例返回了 address_home 集合中的地址数据：

```
{
   "_id" : ObjectId("534009e4d852427820000002"),
   "building" : "22 A, Indiana Apt",
   "pincode" : 123456,
   "city" : "Los Angeles",
   "state" : "California"
}
```

MongoDB 覆盖索引查询

官方的MongoDB的文档中说明，覆盖查询是以下的查询：

- 所有的查询字段是索引的一部分
- 所有的查询返回字段在同一个索引中

由于所有出现在查询中的字段是索引的一部分， MongoDB 无需在整个数据文档中检索匹配查询条件和返回使用相同索引的查询结果。

因为索引存在于RAM中，从索引中获取数据比通过扫描文档读取数据要快得多。

使用覆盖索引查询

为了测试盖索引查询，使用以下 users 集合:

```
{
   "_id": ObjectId("53402597d852426020000002"),
   "contact": "987654321",
   "dob": "01-01-1991",
   "gender": "M",
   "name": "Tom Benzamin",
   "user_name": "tombenzamin"
}
```

我们在 users 集合中创建联合索引，字段为 gender 和 user_name :

```
>db.users.ensureIndex({gender:1,user_name:1})
```

现在，该索引会覆盖以下查询：

```
>db.users.find({gender:"M"},{user_name:1,_id:0})
```

也就是说，对于上述查询，MongoDB的不会去数据库文件中查找。相反， 它会从索引中提取数据，这是非常快速的数据查询。

由于我们的索引中不包括 _id 字段，_id在查询中会默认返回，我们可以在MongoDB的查询结果集中排除它。

下面的实例没有排除_id，查询就不会被覆盖：

```
>db.users.find({gender:"M"},{user_name:1})
```

最后，如果是以下的查询，不能使用覆盖索引查询：

- 所有索引字段是一个数组

MongoDB 查询分析

MongoDB 查询分析可以确保我们建议的索引是否有效，是查询语句性能分析的重要工具。

MongoDB 查询分析常用函数有：explain() 和 hint()。

使用 explain()

explain 操作提供了查询信息，使用索引及查询统计等。有利于我们对索引的优化。

接下来我们在 users 集合中创建 gender 和 user_name 的索引：

```
>db.users.ensureIndex({gender:1,user_name:1})
</p>
<p>现在在查询语句中使用 explain ：</p>
<pre>
>db.users.find({gender:"M"},{user_name:1,_id:0}).explain()
```

以上的 explain() 查询返回如下结果：

```
{
   "cursor" : "BtreeCursor gender_1_user_name_1",
   "isMultiKey" : false,
   "n" : 1,
   "nscannedObjects" : 0,
   "nscanned" : 1,
   "nscannedObjectsAllPlans" : 0,
   "nscannedAllPlans" : 1,
   "scanAndOrder" : false,
   "indexOnly" : true,
   "nYields" : 0,
   "nChunkSkips" : 0,
   "millis" : 0,
   "indexBounds" : {
      "gender" : [
         [
            "M",
            "M"
         ]
      ],
      "user_name" : [
         [
            {
               "$minElement" : 1
            },
            {
               "$maxElement" : 1
            }
         ]
      ]
   }
}
```

现在，我们看看这个结果集的字段：

- **indexOnly**: 字段为 true ，表示我们使用了索引。
- **cursor**：因为这个查询使用了索引，MongoDB中索引存储在B树结构中，所以这是也使用了BtreeCursor类型的游标。如果没有使用索引，游标的类型是BasicCursor。这个键还会给出你所使用的索引的名称，你通过这个名称可以查看当前数据库下的system.indexes集合（系统自动创建，由于存储索引信息，这个稍微会提到）来得到索引的详细信息。
- **n**：当前查询返回的文档数量。
- **nscanned/nscannedObjects**：表明当前这次查询一共扫描了集合中多少个文档，我们的目的是，让这个数值和返回文档的数量越接近越好。
- **millis**：当前查询所需时间，毫秒数。
- **indexBounds**：当前查询具体使用的索引。

使用 hint()

虽然MongoDB查询优化器一般工作的很不错，但是也可以使用hints来强迫MongoDB使用一个指定的索引。

这种方法某些情形下会提升性能。 一个有索引的collection并且执行一个多字段的查询(一些字段已经索引了)。

如下查询实例指定了使用 gender 和 user_name 索引字段来查询：

```
>db.users.find({gender:"M"},{user_name:1,_id:0}).hint({gender:1,use
```

可以使用 explain() 函数来分析以上查询：

```
>db.users.find({gender:"M"},{user_name:1,_id:0}).hint({gender:1,use
```

MongoDB 原子操作

mongodb不支持事务，所以，在你的项目中应用时，要注意这点。无论什么设计，都不要要求mongodb保证数据的完整性。

但是mongodb提供了许多原子操作，比如文档的保存，修改，删除等，都是原子操作。

所谓原子操作就是要么这个文档保存到Mongodb，要么没有保存到Mongodb，不会出现查询到的文档没有保存完整的情况。

原子操作数据模型

考虑下面的例子，图书馆的书籍及结账信息。

实例说明了在一个相同的文档中如何确保嵌入字段关联原子操作（update：更新）的字段是同步的。

```
book = {
          _id: 123456789,
          title: "MongoDB: The Definitive Guide",
          author: [ "Kristina Chodorow", "Mike Dirolf" ],
          published_date: ISODate("2010-09-24"),
          pages: 216,
          language: "English",
          publisher_id: "oreilly",
          available: 3,
          checkout: [ { by: "joe", date: ISODate("2012-10-15") } ]
        }
```

你可以使用 db.collection.findAndModify() 方法来判断书籍是否可结算并更新新的结算信息。

在同一个文档中嵌入的 available 和 checkout 字段来确保这些字段是同步更新的:

```
db.books.findAndModify ( {
   query: {
            _id: 123456789,
            available: { $gt: 0 }
          },
   update: {
             $inc: { available: -1 },
             $push: { checkout: { by: "abc", date: new Date() } }
           }
} )
```

原子操作常用命令

$set

用来指定一个键并更新键值，若键不存在并创建。

```
{ $set : { field : value } }
```

$unset

用来删除一个键。

```
{ $unset : { field : 1} }
```

$inc

$inc可以对文档的某个值为数字型（只能为满足要求的数字）的键进行增减的操作。

```
{ $inc : { field : value } }
```

$push

用法：

```
{ $push : { field : value } }
```

把value追加到field里面去，field一定要是数组类型才行，如果field不存在，会新增一个数组类型加进去。

$pushAll

同$push,只是一次可以追加多个值到一个数组字段内。

```
{ $pushAll : { field : value_array } }
```

$pull

从数组field内删除一个等于value值。

```
{ $pull : { field : _value } }
```

$addToSet

增加一个值到数组内，而且只有当这个值不在数组内才增加。

$pop

删除数组的第一个或最后一个元素

```
{ $pop : { field : 1 } }
```

$rename

修改字段名称

```
{ $rename : { old_field_name : new_field_name } }
```

$bit

位操作，integer类型

```
{$bit : { field : {and : 5}}}
```

偏移操作符

```
> t.find() { "_id" : ObjectId("4b97e62bf1d8c7152c9ccb74"), "title"

> t.update( {'comments.by':'joe'}, {$inc:{'comments.$.votes':1}},

> t.find() { "_id" : ObjectId("4b97e62bf1d8c7152c9ccb74"), "title"
```



MongoDB 高级索引

考虑以下文档集合（users ）：

```
{
   "address": {
      "city": "Los Angeles",
      "state": "California",
      "pincode": "123"
   },
   "tags": [
      "music",
      "cricket",
      "blogs"
   ],
   "name": "Tom Benzamin"
}
```

以上文档包含了 address 子文档和 tags 数组。

索引数组字段

假设我们基于标签来检索用户，为此我们需要对集合中的数组 tags 建立索引。

在数组中创建索引，需要对数组中的每个字段依次建立索引。所以在我们为数组 tags 创建索引时，会为 music、cricket、blogs三个值建立单独的索引。

使用以下命令创建数组索引：

```
>db.users.ensureIndex({"tags":1})
```

创建索引后，我们可以这样检索集合的 tags 字段：

```
>db.users.find({tags:"cricket"})
```

为了验证我们使用使用了索引，可以使用 explain 命令：

```
>db.users.find({tags:"cricket"}).explain()
```

以上命令执行结果中会显示 "cursor" : "BtreeCursor tags_1" ，则表示已经使用了索引。

索引子文档字段

假设我们需要通过city、state、pincode字段来检索文档，由于这些字段是子文档的字段，所以我们需要对子文档建立索引。

为子文档的三个字段创建索引，命令如下：

```
>db.users.ensureIndex({"address.city":1,"address.state":1,"address.
```

一旦创建索引，我们可以使用子文档的字段来检索数据：

```
>db.users.find({"address.city":"Los Angeles"})
```

记住查询表达式必须遵循指定的索引的顺序。所以上面创建的索引将支持以下查询：

```
>db.users.find({"address.city":"Los Angeles","address.state":"Calif
```

同样支持以下查询：

```
>db.users.find({"address.city":"LosAngeles","address.state":"Califo
```

MongoDB 索引限制

额外开销

每个索引占据一定的存储空间，在进行插入，更新和删除操作时也需要对索引进行操作。所以，如果你很少对集合进行读取操作，建议不使用索引。

内存(RAM)使用

由于索引是存储在内存(RAM)中,你应该确保该索引的大小不超过内存的限制。

如果索引的大小大于内存的限制，MongoDB会删除一些索引，这将导致性能下降。

查询限制

索引不能被以下的查询使用：

- 正则表达式及非操作符，如 $nin, $not, 等。
- 算术运算符，如 $mod, 等。
- $where 子句

所以，检测你的语句是否使用索引是一个好的习惯，可以用explain来查看。

索引键限制

从2.6版本开始，如果现有的索引字段的值超过索引键的限制，MongoDB中不会创建索引。

插入文档超过索引键限制

如果文档的索引字段值超过了索引键的限制，MongoDB不会将任何文档转换成索引的集合。与mongorestore和mongoimport工具类似。

最大范围

- 集合中索引不能超过64个
- 索引名的长度不能超过125个字符
- 一个复合索引最多可以有31个字段

MongoDB ObjectId

在前面几个章节中我们已经使用了MongoDB 的对象 Id(ObjectId)。

在本章节中，我们将了解的ObjectId的结构。

ObjectId 是一个12字节 BSON 类型数据，有以下格式：

- 前4个字节表示时间戳
- 接下来的3个字节是机器标识码
- 紧接的两个字节由进程id组成（PID）
- 最后三个字节是随机数。

MongoDB中存储的文档必须有一个"_id"键。这个键的值可以是任何类型的，默认是个ObjectId对象。

在一个集合里面，每个集合都有唯一的"_id"值，来确保集合里面每个文档都能被唯一标识。

MongoDB采用ObjectId，而不是其他比较常规的做法（比如自动增加的主键）的主要原因，因为在多个 服务器上同步自动增加主键值既费力还费时。

创建信的ObjectId

使用以下代码生成新的ObjectId：

```
>newObjectId = ObjectId()
```

上面的语句返回以下唯一生成的id：

```
ObjectId("5349b4ddd2781d08c09890f3")
```

你也可以使用生成的id来取代MongoDB自动生成的ObjectId：

```
>myObjectId = ObjectId("5349b4ddd2781d08c09890f4")
```

创建文档的时间戳

由于 ObjectId 中存储了 4 个字节的时间戳，所以你不需要为你的文档保存时间戳字段，你可以通过 getTimestamp 函数来获取文档的创建时间:

```
>ObjectId("5349b4ddd2781d08c09890f4").getTimestamp()
```

以上代码将返回 ISO 格式的文档创建时间：

```
ISODate("2014-04-12T21:49:17Z")
```

ObjectId 转换为字符串

在某些情况下，您可能需要将ObjectId转换为字符串格式。你可以使用下面的代码：

```
>new ObjectId.str
```

以上代码将返回Guid格式的字符串：：

```
5349b4ddd2781d08c09890f3
```

MongoDB Map Reduce

Map-Reduce是一种计算模型，简单的说就是将大批量的工作（数据）分解（MAP）执行，然后再将结果合并成最终结果（REDUCE）。

MongoDB提供的Map-Reduce非常灵活，对于大规模数据分析也相当实用。

MapReduce 命令

以下是MapReduce的基本语法：

```
>db.collection.mapReduce(
   function() {emit(key,value);},  //map 函数
   function(key,values) {return reduceFunction},   //reduce 函数
   {
      out: collection,
      query: document,
      sort: document,
      limit: number
   }
)
```

使用 MapReduce 要实现两个函数 Map 函数和 Reduce 函数,Map 函数调用 emit(key, value), 遍历 collection 中所有的记录, 将key 与 value 传递给 Reduce 函数进行处理。

Map 函数必须调用 emit(key, value) 返回键值对。

参数说明:

- **map** ：映射函数 (生成键值对序列,作为 reduce 函数参数)。
- **reduce** 统计函数，reduce函数的任务就是将key-values变成key-value，也就是把values数组变成一个单一的值value。。
- **out** 统计结果存放集合 (不指定则使用临时集合,在客户端断开后自动删除)。
- **query** 一个筛选条件，只有满足条件的文档才会调用map函数。（query。limit，sort可以随意组合）
- **sort** 和limit结合的sort排序参数（也是在发往map函数前给文档排序），可以优化分组机制
- **limit** 发往map函数的文档数量的上限（要是没有limit，单独使用sort的用处不大）

使用 MapReduce

考虑以下文档结构存储用户的文章，文档存储了用户的 user_name 和文章的 status字段：

```
{
   "post_text": "w3cschool.cc 菜鸟教程，最全的技术文档。",
   "user_name": "mark",
   "status":"active"
}
```

现在，我们将在 posts 集合中使用 mapReduce 函数来选取已发布的文章，并通过 user_name分组，计算每个用户的文章数：

```
>db.posts.mapReduce(
   function() { emit(this.user_id,1); },
   function(key, values) {return Array.sum(values)},
      {
         query:{status:"active"},
         out:"post_total"
      }
)
```

以上 mapReduce 输出结果为：

```
{
   "result" : "post_total",
   "timeMillis" : 9,
   "counts" : {
      "input" : 4,
      "emit" : 4,
      "reduce" : 2,
      "output" : 2
   },
   "ok" : 1,
}
```

结果表明，共有4个符合查询条件（status:"active"）的文档， 在map函数中生成了4个键值对文档，最后使用reduce函数将相同的键值分为两组。

具体参数说明：

- result：储存结果的collection的名字,这是个临时集合，MapReduce的连接关闭后自动就被删除了。
- timeMillis：执行花费的时间，毫秒为单位
- input：满足条件被发送到map函数的文档个数
- emit：在map函数中emit被调用的次数，也就是所有集合中的数据总量
- ouput：结果集合中的文档个数（**count**对调试非常有帮助）
- ok：是否成功，成功为1

- err：如果失败，这里可以有失败原因，不过从经验上来看，原因比较模糊，作用不大

使用 find 操作符来查看 mapReduce 的查询结果：

```
>db.posts.mapReduce(
   function() { emit(this.user_id,1); },
   function(key, values) {return Array.sum(values)},
      {
         query:{status:"active"},
         out:"post_total"
      }
).find()
```

以上查询显示如下结果，两个用户 tom 和 mark 有两个发布的文章:

```
{ "_id" : "tom", "value" : 2 }
{ "_id" : "mark", "value" : 2 }
```

用类似的方式，MapReduce可以被用来构建大型复杂的聚合查询。

Map函数和Reduce函数可以使用 JavaScript 来实现，是的MapReduce的使用非常灵活和强大。

MongoDB 全文检索

全文检索对每一个词建立一个索引，指明该词在文章中出现的次数和位置，当用户查询时，检索程序就根据事先建立的索引进行查找，并将查找的结果反馈给用户的检索方式。

这个过程类似于通过字典中的检索字表查字的过程。

MongoDB 从 2.4 版本开始支持全文检索，目前支持15种语言(暂时不支持中文)的全文索引。

- danish
- dutch
- english
- finnish
- french
- german
- hungarian
- italian
- norwegian
- portuguese
- romanian
- russian
- spanish
- swedish
- turkish

启用全文检索

MongoDB 在 2.6 版本以后是默认开启全文检索的，如果你使用之前的版本，你需要使用以下代码来启用全文检索:

```
>db.adminCommand({setParameter:true,textSearchEnabled:true})
```

或者使用命令：

```
mongod --setParameter textSearchEnabled=true
```

创建全文索引

考虑以下 posts 集合的文档数据，包含了文章内容（post_text）及标签(tags)：

```
{
   "post_text": "enjoy the mongodb articles on w3cschool.cc",
   "tags": [
      "mongodb",
      "w3cschool"
   ]
}
```

我们可以对 post_text 字段建立全文索引，这样我们可以搜索文章内的内容：

```
>db.posts.ensureIndex({post_text:"text"})
```

使用全文索引

现在我们已经对 post_text 建立了全文索引，我们可以搜索文章中的关键词 w3cschool.cc：

```
>db.posts.find({$text:{$search:"w3cschool.cc"}})
```

以下命令返回了如下包含w3cschool.cc关键词的文档数据：

```
{
   "_id" : ObjectId("53493d14d852429c10000002"),
   "post_text" : "enjoy the mongodb articles on w3cschool.cc",
   "tags" : [ "mongodb", "w3cschool" ]
}
{
   "_id" : ObjectId("53493d1fd852429c10000003"),
   "post_text" : "writing tutorials on w3cschool.cc",
   "tags" : [ "mongodb", "tutorial" ]
}
```

如果你使用的是旧版本的MongoDB，你可以使用以下命令：

```
>db.posts.runCommand("text",{search:" w3cschool.cc"})
```

使用全文索引可以提高搜索效率。

删除全文索引

删除已存在的全文索引，可以使用 find 命令查找索引名：

```
>db.posts.getIndexes()
```

通过以上命令获取索引名，本例的索引名为post_text_text，执行以下命令来删除索引：

```
>db.posts.dropIndex("post_text_text")
```

MongoDB 正则表达式

正则表达式是使用单个字符串来描述、匹配一系列符合某个句法规则的字符串。

许多程序设计语言都支持利用正则表达式进行字符串操作。

MongoDB 使用 **$regex** 操作符来设置匹配字符串的正则表达式。

MongoDB使用PCRE (Perl Compatible Regular Expression) 作为正则表达式语言。

不同于全文检索，我们使用正则表达式不需要做任何配置。

考虑以下 **posts** 集合的文档结构，该文档包含了文章内容和标签：

```
{
   "post_text": "enjoy the mongodb articles on tutorialspoint",
   "tags": [
      "mongodb",
      "tutorialspoint"
   ]
}
```

使用正则表达式

以下命令使用正则表达式查找包含 w3cschool.cc 字符串的文章：

```
>db.posts.find({post_text:{$regex:"w3cschool.cc"}})
```

以上查询也可以写为：

```
>db.posts.find({post_text:/w3cschool.cc/})
```

不区分大小写的正则表达式

如果检索需要不区分大小写，我们可以设置 $options 为 $i。

以下命令将查找不区分大小写的字符串 w3cschool.cc：

```
>db.posts.find({post_text:{$regex:"w3cschool.cc",$options:"$i"}})
```

集合中会返回所有包含字符串 w3cschool.cc 的数据，且不区分大小写：

```
{
   "_id" : ObjectId("53493d37d852429c10000004"),
   "post_text" : "hey! this is my post on  W3Cschool.cc",
   "tags" : [ "tutorialspoint" ]
}
```

数组元素使用正则表达式

我们还可以在数组字段中使用正则表达式来查找内容。 这在标签的实现上非常有用，如果你需要查找包含以 tutorial 开头的标签数据(tutorial 或 tutorials 或 tutorialpoint 或 tutorialphp)， 你可以使用以下代码：

```
>db.posts.find({tags:{$regex:"tutorial"}})
```

优化正则表达式查询

- 如果你的文档中字段设置了索引，那么使用索引相比于正则表达式匹配查找所有的数据查询速度更快。
- 如果正则表达式是前缀表达式，所有匹配的数据将以指定的前缀字符串为开始。例如： 如果正则表达式为 **^tut** ， 查询语句将查找以 tut 为开头的字符串。

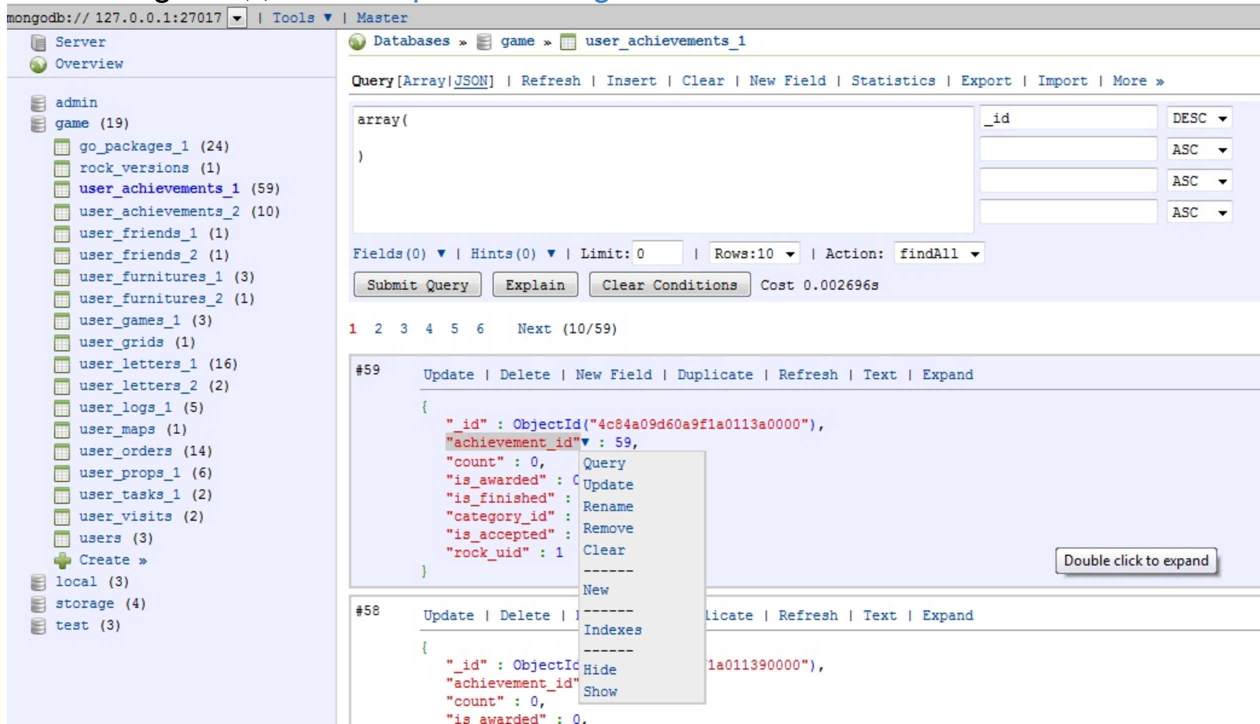
MongoDB 管理工具: Rockmongo

RockMongo是PHP5写的一个MongoDB管理工具。

通过 Rockmongo 你可以管理 MongoDB服务，数据库，集合，文档，索引等等。

它提供了非常人性化的操作。类似 phpMyAdmin（PHP开发的MySql管理工具）。

Rockmongo 下载地址：http://rockmongo.com/downloads



简介

主要特征：

- 使用宽松的New BSD License协议
- 速度快，安装简单
- 支持多语言（目前提供中文、英文、日文、巴西葡萄牙语、法语、德语、俄语、意大利语）
- 系统
    - 可以配置多个主机，每个主机可以有多个管理员
    - 需要管理员密码才能登入操作，确保数据库的安全性
- 服务器
    - 服务器信息 (WEB服务器, PHP, PHP.ini相关指令 ...)
    - 状态
    - 数据库信息
- 数据库
    - 查询，创建和删除
    - 执行命令和Javascript代码

    - 统计信息
- 集合（相当于表）
    - 强大的查询工具
    - 读数据，写数据，更改数据，复制数据，删除数据
    - 查询、创建和删除索引
    - 清空数据
    - 批量删除和更改数据
    - 统计信息
- GridFS
    - 查看分块
    - 下载文件

安装

需求

- 一个能运行PHP的Web服务器，比如Apache Httpd, Nginx ...
- PHP - 需要PHP v5.1.6或更高版本，需要支持SESSION
    - 为了能连接MongoDB，你需要安装php_mongo扩展

快速安装

- 下载安装包
- 解压到你的网站目录下
- 用编辑器打开config.php，修改host, port, admins等参数
- 在浏览器中访问index.php，比如说：http://localhost/rockmongo/index.php
- 使用用户名和密码登录，默认为"admin"和"admin"
- 开始玩转MongoDB!

参考文章：http://rockmongo.com/wiki/introduction?lang=zh_cn

MongoDB GridFS

GridFS 用于存储和恢复那些超过16M（BSON文件限制）的文件(如：图片、音频、视频等)。

GridFS 也是文件存储的一种方式，但是它是存储在MonoDB的集合中。

GridFS 可以更好的存储大于16M的文件。

GridFS 会将大文件对象分割成多个小的chunk(文件片段),一般为256k/个,每个chunk将作为MongoDB的一个文档(document)被存储在chunks集合中。

GridFS 用两个集合来存储一个文件：fs.files与fs.chunks。

每个文件的实际内容被存在chunks(二进制数据)中,和文件有关的meta数据(filename,content_type,还有用户自定义的属性)将会被存在files集合中。

以下是简单的 fs.files 集合文档：

```
{
   "filename": "test.txt",
   "chunkSize": NumberInt(261120),
   "uploadDate": ISODate("2014-04-13T11:32:33.557Z"),
   "md5": "7b762939321e146569b07f72c62cca4f",
   "length": NumberInt(646)
}
```

以下是简单的 fs.chunks 集合文档：

```
{
   "files_id": ObjectId("534a75d19f54bfec8a2fe44b"),
   "n": NumberInt(0),
   "data": "Mongo Binary Data"
}
```

GridFS 添加文件

现在我们使用 GridFS 的 put 命令来存储 mp3 文件。 调用 MongoDB 安装目录下bin的 mongofiles.exe工具。

打开命令提示符，进入到MongoDB的安装目录的bin目录中，找到mongofiles.exe，并输入下面的代码：

```
>mongofiles.exe -d gridfs put song.mp3
```

GridFS 是存储文件的数据名称。如果不存在该数据库，MongoDB会自动创建。Song.mp3 是音频文件名。

使用以下命令来查看数据库中文件的文档：

```
>db.fs.files.find()
```

以上命令执行后返回以下文档数据：

```
{
   _id: ObjectId('534a811bf8b4aa4d33fdf94d'),
   filename: "song.mp3",
   chunkSize: 261120,
   uploadDate: new Date(1397391643474), md5: "e4f53379c909f7bed2e9c
   length: 10401959
}
```

我们可以看到 fs.chunks 集合中所有的区块，以下我们得到了文件的 _id 值，我们可以根据这个 _id 获取区块(chunk)的数据：

```
>db.fs.chunks.find({files_id:ObjectId('534a811bf8b4aa4d33fdf94d')})
```

以上实例中，查询返回了 40 个文档的数据，意味着mp3文件被存储在40个区块中。

MongoDB 固定集合（Capped Collections）

MongoDB 固定集合（Capped Collections）是性能出色的有着固定大小的集合，对于大小固定，我们可以想象其就像一个环形队列，当集合空间用完后，再插入的元素就会覆盖最初始的头部的元素！

创建固定集合

我们通过createCollection来创建一个固定集合，且capped选项设置为true：

```
>db.createCollection("cappedLogCollection",{capped:true,size:10000]
```

还可以指定文档个数,加上max:1000属性：

```
>db.createCollection("cappedLogCollection",{capped:true,size:10000,
```

判断集合是否为固定集合:

```
>db.cappedLogCollection.isCapped()
```

如果需要将已存在的集合转换为固定集合可以使用以下命令：

```
>db.runCommand({"convertToCapped":"posts",size:10000})
```

以上代码将我们已存在的 posts 集合转换为固定集合。

固定集合查询

固定集合文档按照插入顺序储存的,默认情况下查询就是按照插入顺序返回的,也可以使用$natural调整返回顺序。

```
>db.cappedLogCollection.find().sort({$natural:-1})
```

固定集合的功能特点

可以插入及更新,但更新不能超出collection的大小,否则更新失败,不允许删除,但是可以调用drop()删除集合中的所有行,但是drop后需要显式地重建集合。

在32位机子上一个cappped collection的最大值约为482.5M,64位上只受系统文件大小的限制。

固定集合属性及用法

属性

- 属性1:对固定集合进行插入速度极快
- 属性2:按照插入顺序的查询输出速度极快
- 属性3:能够在插入最新数据时,淘汰最早的数据

用法

- 用法1:储存日志信息
- 用法2:缓存一些少量的文档

MongoDB 自动增长

MongoDB 没有像 SQL 一样有自动增长的功能， MongoDB 的 _id 是系统自动生成的12字节唯一标识。

但在某些情况下，我们可以需要实现 ObjectId 实现自动增长功能。

由于 MongoDB 没有实现这个功能，我们可以通过编程的方式来实现，以下我们将在 counters 集合中实现_id字段自动增长。

使用 counters 集合

考虑以下 products 文档。我们希望 _id 字段实现 从 1,2,3,4 到 n 的自动增长功能。

```
{
  "_id":1,
  "product_name": "Apple iPhone",
  "category": "mobiles"
}
```

为此，创建 counters 集合，序列字段值可以实现自动长：

```
>db.createCollection("counters")
```

现在我们向 counters 集合中插入以下文档，使用 productid 作为 key:

```
{
  "_id":"productid",
  "sequence_value": 0
}
```

sequence_value 字段是序列的是通过自动增长后的一个值。

使用以下命令插入 counters 集合的序列文档中：

```
>db.counters.insert({_id:"productid",sequence_value:0})
```

创建 Javascript 函数

现在，我们创建函数 getNextSequenceValue 来作为序列名的输入， 指定的序列会自动增长 1 并返回最新序列值。在本文的实例中序列名为 productid 。

```
>function getNextSequenceValue(sequenceName){
   var sequenceDocument = db.counters.findAndModify(
      {
         query:{_id: sequenceName },
         update: {$inc:{sequence_value:1}},
         new:true
      });
   return sequenceDocument.sequence_value;
}
```

使用 Javascript 函数

接下来我们将使用 getNextSequenceValue 函数创建一个新的文档， 并设置文档 _id 自动为返回的序列值：

```
>db.products.insert({
   "_id":getNextSequenceValue("productid"),
   "product_name":"Apple iPhone",
   "category":"mobiles"})

>db.products.insert({
   "_id":getNextSequenceValue("productid"),
   "product_name":"Samsung S3",
   "category":"mobiles"})
```

就如你所看到的，我们使用 getNextSequenceValue 函数来设置 _id 字段。

为了验证函数是否有效，我们可以使用以下命令读取文档：

```
>db.prodcuts.find()
```

以上命令将返回以下结果，我们发现 _id 字段是自增长的：

```
{ "_id" : 1, "product_name" : "Apple iPhone", "category" : "mobiles

{ "_id" : 2, "product_name" : "Samsung S3", "category" : "mobiles"
```

W3School Redis 教程 & 命令参考

来源：

- Redis 教程
- Redis 命令参考

整理：飞龙

Redis 教程

Redis 简介

Redis 是完全开源免费的，遵守BSD协议，是一个高性能的key-value数据库。

Redis 与其他 key - value 缓存产品有以下三个特点：

- Redis支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用。
- Redis不仅仅支持简单的key-value类型的数据，同时还提供list，set，zset，hash等数据结构的存储。
- Redis支持数据的备份，即master-slave模式的数据备份。

Redis 优势

- 性能极高 – Redis能读的速度是110000次/s,写的速度是81000次/s 。
- 丰富的数据类型 – Redis支持二进制案例的 Strings, Lists, Hashes, Sets 及 Ordered Sets 数据类型操作。
- 原子 – Redis的所有操作都是原子性的，同时Redis还支持对几个操作全并后的原子性执行。
- 丰富的特性 – Redis还支持 publish/subscribe, 通知, key 过期等等特性。

Redis与其他key-value存储有什么不同？

- Redis有着更为复杂的数据结构并且提供对他们的原子性操作，这是一个不同于其他数据库的进化路径。Redis的数据类型都是基于基本数据结构的同时对程序员透明，无需进行额外的抽象。
- Redis运行在内存中但是可以持久化到磁盘，所以在对不同数据集进行高速读写时需要权衡内存，应为数据量不能大于硬件内存。在内存数据库方面的另一个优点是， 相比在磁盘上相同的复杂的数据结构，在内存中操作起来非常简单，这样Redis可以做很多内部复杂性很强的事情。 同时，在磁盘格式方面他们是紧凑的以追加的方式产生的，因为他们并不需要进行随机访问。

Redis 安装

Window 下安装

下载地址：https://github.com/dmajkic/redis/downloads。

下载到的Redis支持32bit和64bit。根据自己实际情况选择，将64bit的内容cp到自定义盘符安装目录取名redis。 如 C:\reids

打开一个cmd窗口 使用cd命令切换目录到 C:\redis 运行 **redis-server.exe redis.conf** 。

如果想方便的话，可以把redis的路径加到系统的环境变量里，这样就省得再输路径了，后面的那个redis.conf可以省略，如果省略，会启用默认的。输入之后，会显示如下界面：

```
C:\WINDOWS\system32\cmd.exe - redis-server.exe redis.conf
C:\redis>redis-server.exe redis.conf
[7468] 13 Sep 14:36:13 * Server started, Redis version 2.4.5
[7468] 13 Sep 14:36:13 * DB loaded from disk: 0 seconds
[7468] 13 Sep 14:36:13 * The server is now ready to accept connections on port 6
379
[7468] 13 Sep 14:36:14 - DB 0: 1 keys (0 volatile) in 4 slots HT.
[7468] 13 Sep 14:36:14 - 0 clients connected (0 slaves), 672860 bytes in use
[7468] 13 Sep 14:36:20 - DB 0: 1 keys (0 volatile) in 4 slots HT.
[7468] 13 Sep 14:36:20 - 0 clients connected (0 slaves), 672860 bytes in use
[7468] 13 Sep 14:36:25 - DB 0: 1 keys (0 volatile) in 4 slots HT.
[7468] 13 Sep 14:36:25 - 0 clients connected (0 slaves), 672860 bytes in use
[7468] 13 Sep 14:36:31 - DB 0: 1 keys (0 volatile) in 4 slots HT.
[7468] 13 Sep 14:36:31 - 0 clients connected (0 slaves), 672860 bytes in use
[7468] 13 Sep 14:36:36 - DB 0: 1 keys (0 volatile) in 4 slots HT.
[7468] 13 Sep 14:36:36 - 0 clients connected (0 slaves), 672860 bytes in use
[7468] 13 Sep 14:36:42 - DB 0: 1 keys (0 volatile) in 4 slots HT.
[7468] 13 Sep 14:36:42 - 0 clients connected (0 slaves), 672860 bytes in use
[7468] 13 Sep 14:36:47 - DB 0: 1 keys (0 volatile) in 4 slots HT.
[7468] 13 Sep 14:36:47 - 0 clients connected (0 slaves), 672860 bytes in use
[7468] 13 Sep 14:36:52 - DB 0: 1 keys (0 volatile) in 4 slots HT.
[7468] 13 Sep 14:36:52 - 0 clients connected (0 slaves), 672860 bytes in use
[7468] 13 Sep 14:36:58 - DB 0: 1 keys (0 volatile) in 4 slots HT.
[7468] 13 Sep 14:36:58 - 0 clients connected (0 slaves), 672860 bytes in use
[7468] 13 Sep 14:37:03 - DB 0: 1 keys (0 volatile) in 4 slots HT.
[7468] 13 Sep 14:37:03 - 0 clients connected (0 slaves), 672860 bytes in use
```

这时候别启一个cmd窗口，原来的不要关闭，不然就无法访问服务端了。

切换到redis目录下运行 **redis-cli.exe -h 127.0.0.1 -p 6379** 。

设置键值对 **set myKey abc**

取出键值对 **get myKey**

```
C:\WINDOWS\system32\cmd.exe - redis-cli.exe -h 127.0.0.1 -p 6379
C:\redis>redis-cli.exe -h 127.0.0.1 -p 6379
redis 127.0.0.1:6379>
redis 127.0.0.1:6379> set myKey abc
OK
redis 127.0.0.1:6379> get myKey
"abc"
redis 127.0.0.1:6379>
```

Linux 下安装

下载地址：http://redis.io/download，下载最新文档版本。

本教程使用的最新文档版本为 2.8.17，下载并安装：

```
$ wget http://download.redis.io/releases/redis-2.8.17.tar.gz
$ tar xzf redis-2.8.17.tar.gz
$ cd redis-2.8.17
$ make
```

make完后 redis-2.8.17目录下会出现编译后的redis服务程序redis-server,还有用于测试的客户端程序redis-cli

下面启动redis服务.

```
$ ./redis-server
```

注意这种方式启动redis 使用的是默认配置。也可以通过启动参数告诉redis使用指定配置文件使用下面命令启动。

```
$ ./redis-server redis.conf
```

redis.conf是一个默认的配置文件。我们可以根据需要使用自己的配置文件。

启动redis服务进程后，就可以使用测试客户端程序redis-cli和redis服务交互了。 比如：

```
$ ./redis-cli
redis> set foo bar
OK
redis> get foo
"bar"
```

Ubuntu 下安装

在 Ubuntu 系统安装 Redi 可以使用以下命令:

```
$sudo apt-get update
$sudo apt-get install redis-server
```

启动 Redis

```
$redis-server
```

查看 redis 是否启动？

```
$redis-cli
```

以上命令将打开以下终端：

```
redis 127.0.0.1:6379>
```

127.0.0.1 是本机 IP，6379 是 redis 服务端口。现在我们输入 PING 命令。

```
redis 127.0.0.1:6379> ping
PONG
```

以上说明我们已经成功安装了redis。

Redis 配置

Redis 的配置文件位于 Redis 安装目录下，文件名为 redis.conf。

你可以通过 **CONFIG** 命令查看或设置配置项。

语法

Redis CONFIG 命令格式如下：

```
redis 127.0.0.1:6379&gt; CONFIG GET CONFIG_SETTING_NAME
```

实例

```
redis 127.0.0.1:6379> CONFIG GET loglevel

1) "loglevel"
2) "notice"
```

使用 * 号获取所有配置项：

实例

```
redis 127.0.0.1:6379> CONFIG GET *

  1) "dbfilename"
  2) "dump.rdb"
  3) "requirepass"
  4) ""
  5) "masterauth"
  6) ""
  7) "unixsocket"
  8) ""
  9) "logfile"
 10) ""
 11) "pidfile"
 12) "/var/run/redis.pid"
 13) "maxmemory"
 14) "0"
 15) "maxmemory-samples"
 16) "3"
 17) "timeout"
 18) "0"
```

```
19) "tcp-keepalive"
20) "0"
21) "auto-aof-rewrite-percentage"
22) "100"
23) "auto-aof-rewrite-min-size"
24) "67108864"
25) "hash-max-ziplist-entries"
26) "512"
27) "hash-max-ziplist-value"
28) "64"
29) "list-max-ziplist-entries"
30) "512"
31) "list-max-ziplist-value"
32) "64"
33) "set-max-intset-entries"
34) "512"
35) "zset-max-ziplist-entries"
36) "128"
37) "zset-max-ziplist-value"
38) "64"
39) "hll-sparse-max-bytes"
40) "3000"
41) "lua-time-limit"
42) "5000"
43) "slowlog-log-slower-than"
44) "10000"
45) "latency-monitor-threshold"
46) "0"
47) "slowlog-max-len"
48) "128"
49) "port"
50) "6379"
51) "tcp-backlog"
52) "511"
53) "databases"
54) "16"
55) "repl-ping-slave-period"
56) "10"
57) "repl-timeout"
58) "60"
59) "repl-backlog-size"
60) "1048576"
61) "repl-backlog-ttl"
62) "3600"
63) "maxclients"
64) "4064"
65) "watchdog-period"
66) "0"
67) "slave-priority"
68) "100"
69) "min-slaves-to-write"
70) "0"
71) "min-slaves-max-lag"
```

```
 72) "10"
 73) "hz"
 74) "10"
 75) "no-appendfsync-on-rewrite"
 76) "no"
 77) "slave-serve-stale-data"
 78) "yes"
 79) "slave-read-only"
 80) "yes"
 81) "stop-writes-on-bgsave-error"
 82) "yes"
 83) "daemonize"
 84) "no"
 85) "rdbcompression"
 86) "yes"
 87) "rdbchecksum"
 88) "yes"
 89) "activerehashing"
 90) "yes"
 91) "repl-disable-tcp-nodelay"
 92) "no"
 93) "aof-rewrite-incremental-fsync"
 94) "yes"
 95) "appendonly"
 96) "no"
 97) "dir"
 98) "/home/deepak/Downloads/redis-2.8.13/src"
 99) "maxmemory-policy"
100) "volatile-lru"
101) "appendfsync"
102) "everysec"
103) "save"
104) "3600 1 300 100 60 10000"
105) "loglevel"
106) "notice"
107) "client-output-buffer-limit"
108) "normal 0 0 0 slave 268435456 67108864 60 pubsub 33554432 8388
109) "unixsocketperm"
110) "0"
111) "slaveof"
112) ""
113) "notify-keyspace-events"
114) ""
115) "bind"
116) ""
```

编辑配置

你可以通过修改 redis.conf 文件或使用 **CONFIG set** 命令来修改配置。

语法

CONFIG SET 命令基本语法：

```
redis 127.0.0.1:6379> CONFIG SET CONFIG_SETTING_NAME NEW_CONFIG_VAL
```

实例

```
redis 127.0.0.1:6379> CONFIG SET loglevel "notice"
OK
redis 127.0.0.1:6379> CONFIG GET loglevel

1) "loglevel"
2) "notice"
```

参数说明

redis.conf 配置项说明如下：

1. Redis默认不是以守护进程的方式运行，可以通过该配置项修改，使用yes启用守护进程

```
**daemonize no**
```

2. 当Redis以守护进程方式运行时，Redis默认会把pid写入/var/run/redis.pid文件，可以通过pidfile指定

```
**pidfile /var/run/redis.pid**
```

3. 指定Redis监听端口，默认端口为6379，作者在自己的一篇博文中解释了为什么选用6379作为默认端口，因为6379在手机按键上MERZ对应的号码，而MERZ取自意大利歌女Alessia Merz的名字

```
**port 6379**
```

4. 绑定的主机地址

```
**bind 127.0.0.1**
```

5.当 客户端闲置多长时间后关闭连接，如果指定为0，表示关闭该功能

```
**timeout 300**
```

6. 指定日志记录级别，Redis总共支持四个级别：debug、verbose、notice、warning，默认为verbose

```
**loglevel verbose**
```

7. 日志记录方式，默认为标准输出，如果配置Redis为守护进程方式运行，而这里又配置为日志记录方式为标准输出，则日志将会发送给/dev/null

```
**logfile stdout**
```

8. 设置数据库的数量，默认数据库为0，可以使用SELECT <dbid>命令在连接上指定数据库id

```
**databases 16**
```

9. 指定在多长时间内，有多少次更新操作，就将数据同步到数据文件，可以多个条件配合

```
**save &lt;seconds&gt; &lt;changes&gt;**

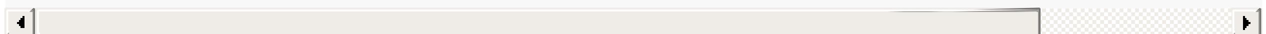
Redis默认配置文件中提供了三个条件：

**save 900 1**
```

save 300 10

save 60 10000

```
分别表示900秒（15分钟）内有1个更改，300秒（5分钟）内有10个更改以及60秒内有10(
```



10. 指定存储至本地数据库时是否压缩数据，默认为yes，Redis采用LZF压缩，如果为了节省CPU时间，可以关闭该选项，但会导致数据库文件变的巨大

```
**rdbcompression yes**
```

11. 指定本地数据库文件名，默认值为dump.rdb

```
**dbfilename dump.rdb**
```

12. 指定本地数据库存放目录

```
**dir ./**
```

13. 设置当本机为slav服务时，设置master服务的IP地址及端口，在Redis启动时，它会自动从master进行数据同步

```
**slaveof &lt;masterip&gt; &lt;masterport&gt;**
```

14. 当master服务设置了密码保护时，slav服务连接master的密码

```
**masterauth &lt;master-password&gt;**
```

15. 设置Redis连接密码，如果配置了连接密码，客户端在连接Redis时需要通过AUTH <password>命令提供密码，默认关闭

```
**requirepass foobared**
```

16. 设置同一时间最大客户端连接数，默认无限制，Redis可以同时打开的客户端连接数为Redis进程可以打开的最大文件描述符数，如果设置 maxclients 0，表示不作限制。当客户端连接数到达限制时，Redis会关闭新的连接并向客户端返回max number of clients reached错误信息

```
**maxclients 128**
```

17. 指定Redis最大内存限制，Redis在启动时会把数据加载到内存中，达到最大内存后，Redis会先尝试清除已到期或即将到期的Key，当此方法处理 后，仍然到达最大内存设置，将无法再进行写入操作，但仍然可以进行读取操作。Redis新的vm机制，会把Key存放内存，Value会存放在swap区

```
**maxmemory &lt;bytes&gt;**
```

18. 指定是否在每次更新操作后进行日志记录，Redis在默认情况下是异步的把数据写入磁盘，如果不开启，可能会在断电时导致一段时间内的数据丢失。因为 redis本身同步数据文件是按上面save条件来同步的，所以有的数据会在一段时间内只存在于内存中。默认为no

```
**appendonly no**
```

19. 指定更新日志文件名，默认为appendonly.aof

```
**appendfilename appendonly.aof**
```

20. 指定更新日志条件，共有3个可选值：
no：表示等操作系统进行数据缓存同步到磁盘（快）
always：表示每次更新操作后手动调用fsync()将数据写到磁盘（慢，安全）
everysec：表示每秒同步一次（折衷，默认值）

```
**appendfsync everysec**
```

21. 指定是否启用虚拟内存机制，默认值为no，简单的介绍一下，VM机制将数据分页存放，由Redis将访问量较少的页即冷数据swap到磁盘上，访问多的页面由磁盘自动换出到内存中（在后面的文章我会仔细分析Redis的VM机制）

```
**vm-enabled no**
```

22. 虚拟内存文件路径，默认值为/tmp/redis.swap，不可多个Redis实例共享

```
**vm-swap-file /tmp/redis.swap**
```

23. 将所有大于vm-max-memory的数据存入虚拟内存,无论vm-max-memory设置多小,所有索引数据都是内存存储的(Redis的索引数据 就是keys),也就是说,当vm-max-memory设置为0的时候,其实是所有value都存在于磁盘。默认值为0

```
**vm-max-memory 0**
```

24. Redis swap文件分成了很多的page，一个对象可以保存在多个page上面，但一个page上不能被多个对象共享，vm-page-size是要根据存储的 数据大小来设定的，作者建议如果存储很多小对象，page大小最好设置为32或者64bytes；如果存储很大大对象，则可以使用更大的page，如果不 确定，就使用默认值

```
**vm-page-size 32**
```

25. 设置swap文件中的page数量，由于页表（一种表示页面空闲或使用的bitmap）是在放在内存中的，，在磁盘上每8个pages将消耗1byte的内存。

```
**vm-pages 134217728**
```

26. 设置访问swap文件的线程数,最好不要超过机器的核数,如果设置为0,那么所有对swap文件的操作都是串行的，可能会造成比较长时间的延迟。默认值为4

```
**vm-max-threads 4**
```

27. 设置在向客户端应答时，是否把较小的包合并为一个包发送，默认为开启

```
**glueoutputbuf yes**
```

28. 指定在超过一定的数量或者最大的元素超过某一临界值时，采用一种特殊的哈希算法

```
**hash-max-zipmap-entries 64**
```

hash-max-zipmap-value 512

29. 指定是否激活重置哈希，默认为开启（后面在介绍Redis的哈希算法时具体介绍）

```
**activerehashing yes**
```

30. 指定包含其它的配置文件，可以在同一主机上多个Redis实例之间使用同一份配置文件，而同时各个实例又拥有自己的特定配置文件

```
**include /path/to/local.conf**
```

Redis 数据类型

Redis支持五种数据类型：string（字符串），hash（哈希），list（列表），set（集合）及zset(sorted set：有序集合)。

String（字符串）

string是redis最基本的类型，你可以理解成与Memcached一模一样的类型，一个key对应一个value。

string类型是二进制安全的。意思是redis的string可以包含任何数据。比如jpg图片或者序列化的对象 。

string类型是Redis最基本的数据类型，一个键最大能存储512MB。

实例

```
redis 127.0.0.1:6379> SET name "w3cschool.cc"
OK
redis 127.0.0.1:6379> GET name
"w3cschool.cc"
```

在以上实例中我们使用了 Redis 的 **SET** 和 **GET** 命令。键为 name，对应的值为 w3cschool.cc。

注意：一个键最大能存储512MB。

Hash（哈希）

Redis hash 是一个键值对集合。

Redis hash是一个string类型的field和value的映射表，hash特别适合用于存储对象。

实例

```
redis 127.0.0.1:6379> HMSET user:1 username w3cschool.cc password w
OK
redis 127.0.0.1:6379> HGETALL user:1
1) "username"
2) "w3cschool.cc"
3) "password"
4) "w3cschool.cc"
5) "points"
6) "200"
redis 127.0.0.1:6379>
```

以上实例中 hash 数据类型存储了包含用户脚本信息的用户对象。 实例中我们使用了 Redis **HMSET, HEGTALL** 命令，**user:1** 为键值。

每个 hash 可以存储 $2^{32 - 1}$ 键值对（40多亿）。

List（列表）

Redis 的 list 类型其实就是一个每个子元素都是string类型的双向链表。你可以将 元素添加到列表的头和尾。

实例

```
redis 127.0.0.1:6379> lpush w3cschool.cc redis
(integer) 1
redis 127.0.0.1:6379> lpush w3cschool.cc mongodb
(integer) 2
redis 127.0.0.1:6379> lpush w3cschool.cc rabitmq
(integer) 3
redis 127.0.0.1:6379> lrange w3cschool.cc 0 10
1) "rabitmq"
2) "mongodb"
3) "redis"
redis 127.0.0.1:6379>
```

列表最多可存储 2^{32 - 1} 元素 (4294967295, 每个列表可存储40多亿)。

set（集合）

Redis的set是string类型的无序集合。

set的是通过hash table实现的，所以添加，删除，查找的复杂度都是O(1)。

sadd 命令

添加一个string元素到,key对应的set集合中，成功返回1,如果元素以及在集合中返回0,key对应的set不存在返回错误。

```
sadd key member
```

实例

```
redis 127.0.0.1:6379> sadd w3cschool.cc redis
(integer) 1
redis 127.0.0.1:6379> sadd w3cschool.cc mongodb
(integer) 1
redis 127.0.0.1:6379> sadd w3cschool.cc rabitmq
(integer) 1
redis 127.0.0.1:6379> sadd w3cschool.cc rabitmq
(integer) 0
redis 127.0.0.1:6379> smembers w3cschool.cc

1) "rabitmq"
2) "mongodb"
3) "redis"
```

注意：以上实例中 rabitmq 添加了两次，但根据集合内元素的唯一性，第二次插入的元素将被忽略。

集合中最大的成员数为 2^{32 - 1} (4294967295, 每个集合可存储40多亿个成员)。

zset(sorted set：有序集合)

Redis zset 和 set 一样也是string类型元素的集合,且不允许重复的成员。

不同的是每个元素都会关联一个double类型的score。redis正是通过分数来为集合中的成员进行从小到大的排序。

zset的成员是唯一的,但分数(score)却可以重复。

zadd 命令

添加元素到集合，元素在集合中存在则更新对应score

```
zadd key score member
```

实例

```
redis 127.0.0.1:6379> zadd w3cschool.cc 0 redis
(integer) 1
redis 127.0.0.1:6379> zadd w3cschool.cc 0 mongodb
(integer) 1
redis 127.0.0.1:6379> zadd w3cschool.cc 0 rabitmq
(integer) 1
redis 127.0.0.1:6379> zadd w3cschool.cc 0 rabitmq
(integer) 0
redis 127.0.0.1:6379> ZRANGEBYSCORE w3cschool.cc 0 1000

1) "redis"
2) "mongodb"
3) "rabitmq"
```

Redis 命令

Redis 命令用于在 redis 服务上执行操作。

要在 redis 服务上执行命令需要一个 redis 客户端。Redis 客户端在我们之前下载的的 redis 的安装包中。

语法

Redis 客户端的基本语法为：

```
$ redis-cli
```

实例

以下实例讲解了如何启动 redis 客户端：

启动 redis 客户端，打开终端并输入命令 **redis-cli**。该命令会连接本地的 redis 服务。

```
$redis-cli
redis 127.0.0.1:6379>
redis 127.0.0.1:6379> PING

PONG
```

在以上实例中我们连接到本地的 redis 服务并执行 **PING** 命令，该命令用于检测 redis 服务是否启动。

在远程服务上执行命令

如果需要在远程 redis 服务上执行命令，同样我们使用的也是 **redis-cli** 命令。

语法

```
$ redis-cli -h host -p port -a password
```

实例

以下实例演示了如何连接到主机为 127.0.0.1，端口为 6379，密码为 mypass 的 redis 服务上。

```
$redis-cli -h 127.0.0.1 -p 6379 -a "mypass"
redis 127.0.0.1:6379>
redis 127.0.0.1:6379> PING

PONG
```

Redis 数据备份与恢复

Redis **SAVE** 命令用于创建当前数据库的备份。

语法

redis Save 命令基本语法如下：

```
redis 127.0.0.1:6379> SAVE
```

实例

```
redis 127.0.0.1:6379> SAVE
OK
```

该命令将在 redis 安装目录中创建dump.rdb文件。

恢复数据

如果需要恢复数据，只需将备份文件 (dump.rdb) 移动到 redis 安装目录并启动服务即可。获取 redis 目录可以使用 **CONFIG** 命令，如下所示：

```
redis 127.0.0.1:6379> CONFIG GET dir
1) "dir"
2) "/usr/local/redis/bin"
```

以上命令 **CONFIG GET dir** 输出的 redis 安装目录为 /usr/local/redis/bin。

Bgsave

创建 redis 备份文件也可以使用命令 **BGSAVE**，该命令在后台执行。

实例

```
127.0.0.1:6379> BGSAVE

Background saving started
```

Redis 安全

我们可以通过 redis 的配置文件设置密码参数，这样客户端连接到 redis 服务就需要密码验证，这样可以让你的 redis 服务更安全。

实例

我们可以通过以下命令查看是否设置了密码验证：

```
127.0.0.1:6379> CONFIG get requirepass
1) "requirepass"
2) ""
```

默认情况下 requirepass 参数是空的，这就意味着你无需通过密码验证就可以连接到 redis 服务。

你可以通过以下命令来修改该参数：

```
127.0.0.1:6379> CONFIG set requirepass "w3cschool.cc"
OK
127.0.0.1:6379> CONFIG get requirepass
1) "requirepass"
2) "w3cschool.cc"
```

设置密码后，客户端连接 redis 服务就需要密码验证，否则无法执行命令。

语法

AUTH 命令基本语法格式如下：

```
127.0.0.1:6379> AUTH password
```

实例

```
127.0.0.1:6379> AUTH "w3cschool.cc"
OK
127.0.0.1:6379> SET mykey "Test value"
OK
127.0.0.1:6379> GET mykey
"Test value"
```

Redis 性能测试

Redis 性能测试是通过同时执行多个命令实现的。

语法

redis 性能测试的基本命令如下：

```
redis-benchmark [option] [option value]
```

实例

以下实例同时执行 10000 个请求来检测性能：

```
redis-benchmark -n 100000

PING_INLINE: 141043.72 requests per second
PING_BULK: 142857.14 requests per second
SET: 141442.72 requests per second
GET: 145348.83 requests per second
INCR: 137362.64 requests per second
LPUSH: 145348.83 requests per second
LPOP: 146198.83 requests per second
SADD: 146198.83 requests per second
SPOP: 149253.73 requests per second
LPUSH (needed to benchmark LRANGE): 148588.42 requests per second
LRANGE_100 (first 100 elements): 58411.21 requests per second
LRANGE_300 (first 300 elements): 21195.42 requests per second
LRANGE_500 (first 450 elements): 14539.11 requests per second
LRANGE_600 (first 600 elements): 10504.20 requests per second
MSET (10 keys): 93283.58 requests per second
```

redis 性能测试工具可选参数如下所示：

序号	选项	描述	默认值
1	-h	指定服务器主机名	127.0.0.1
2	-p	指定服务器端口	6379
3	-s	指定服务器 socket	
4	-c	指定并发连接数	50
5	-n	指定请求数	10000
6	-d	以字节的形式指定 SET/GET 值的数据大小	2
7	-k	1=keep alive 0=reconnect	1
8	-r	SET/GET/INCR 使用随机 key, SADD 使用随机值	
9	-P	通过管道传输 `<numreq>` 请求	1
10	-q	强制退出 redis。仅显示 query/sec 值	
11	--csv	以 CSV 格式输出	
12	-l	生成循环，永久执行测试	
13	-t	仅运行以逗号分隔的测试命令列表。	
14	-I	Idle 模式。仅打开 N 个 idle 连接并等待。	

实例

以下实例我们使用了多个参数来测试 redis 性能：

```
redis-benchmark -h 127.0.0.1 -p 6379 -t set,lpush -n 100000 -q

SET: 146198.83 requests per second
LPUSH: 145560.41 requests per second
```

以上实例中主机为 127.0.0.1，端口号为 6379，执行的命令为 set,lpush，请求数为 10000，通过 -q 参数让结果只显示每秒执行的请求数。

Redis 客户端连接

Redis 通过监听一个 TCP 端口或者 Unix socket 的方式来接收来自客户端的连接，当一个连接建立后，Redis 内部会进行以下一些操作：

- 首先，客户端 socket 会被设置为非阻塞模式，因为 Redis 在网络事件处理上采用的是非阻塞多路复用模型。
- 然后为这个 socket 设置 TCP_NODELAY 属性，禁用 Nagle 算法
- 然后创建一个可读的文件事件用于监听这个客户端 socket 的数据发送

最大连接数

在 Redis2.4 中，最大连接数是被直接硬编码在代码里面的，而在2.6版本中这个值变成可配置的。

maxclients 的默认值是 10000，你也可以在 redis.conf 中对这个值进行修改。

```
config get maxclients

1) "maxclients"
2) "10000"
```

实例

以下实例我们在服务启动时设置最大连接数为 100000：

```
redis-server --maxclients 100000
```

客户端命令

命令	描述
CLIENT LIST	返回连接到 redis 服务的客户端列表
CLIENT SETNAME	设置当前连接的名称
CLIENT GETNAME	获取通过 CLIENT SETNAME 命令设置的服务名称
CLIENT PAUSE	挂起客户端连接，指定挂起的时间以毫秒计
CLIENT KILL	关闭客户端连接

Redis 管道技术

Redis是一种基于客户端-服务端模型以及请求/响应协议的TCP服务。这意味着通常情况下一个请求会遵循以下步骤：

- 客户端向服务端发送一个查询请求，并监听Socket返回，通常是以阻塞模式，等待服务端响应。
- 服务端处理命令，并将结果返回给客户端。

Redis 管道技术

Redis 管道技术可以在服务端未响应时，客户端可以继续向服务端发送请求，并最终一次性读取所有服务端的响应。

实例

查看 redis 管道，只需要启动 redis 实例并输入以下命令：

```
$(echo -en "PING\r\n SET w3ckey redis\r\nGET w3ckey\r\nINCR visitor

+PONG
+OK
redis
:1
:2
:3
```

以上实例中我们通过使用 **PING** 命令查看redis服务是否可用， 之后我们们设置了 w3ckey 的值为 redis，然后我们获取 w3ckey 的值并使得 visitor 自增 3 次。

在返回的结果中我们可以看到这些命令一次性向 redis 服务提交，并最终一次性读取所有服务端的响应

管道技术的优势

管道技术最显著的优势是提高了 redis 服务的性能。

一些测试数据

在下面的测试中，我们将使用Redis的Ruby客户端，支持管道技术特性，测试管道技术对速度的提升效果。

```
require 'rubygems'
require 'redis'
def bench(descr)
start = Time.now
yield
puts "#{descr} #{Time.now-start} seconds"
end
def without_pipelining
r = Redis.new
10000.times {
    r.ping
}
end
def with_pipelining
r = Redis.new
r.pipelined {
    10000.times {
        r.ping
    }
}
end
bench("without pipelining") {
    without_pipelining
}
bench("with pipelining") {
    with_pipelining
}
```

从处于局域网中的Mac OS X系统上执行上面这个简单脚本的数据表明，开启了管道操作后，往返时延已经被改善得相当低了。

```
without pipelining 1.185238 seconds
with pipelining 0.250783 seconds
```

如你所见，开启管道后，我们的速度效率提升了5倍。

Redis 分区

分区是分割数据到多个Redis实例的处理过程，因此每个实例只保存key的一个子集。

分区的优势

- 通过利用多台计算机内存的和值，允许我们构造更大的数据库。
- 通过多核和多台计算机，允许我们扩展计算能力；通过多台计算机和网络适配器，允许我们扩展网络带宽。

分区的不足

redis的一些特性在分区方面表现的不是很好：

- 涉及多个key的操作通常是不被支持的。举例来说，当两个set映射到不同的redis实例上时，你就不能对这两个set执行交集操作。
- 涉及多个key的redis事务不能使用。
- 当使用分区时，数据处理较为复杂，比如你需要处理多个rdb/aof文件，并且从多个实例和主机备份持久化文件。
- 增加或删除容量也比较复杂。redis集群大多数支持在运行时增加、删除节点的透明数据平衡的能力，但是类似于客户端分区、代理等其他系统则不支持这项特性。然而，一种叫做presharding的技术对此是有帮助的。

分区类型

Redis 有两种类型分区。 假设有4个Redis实例 R0，R1，R2，R3，和类似user:1，user:2这样的表示用户的多个key，对既定的key有多种不同方式来选择这个key存放在哪个实例中。也就是说，有不同的系统来映射某个key到某个Redis服务。

范围分区

最简单的分区方式是按范围分区，就是映射一定范围的对象到特定的Redis实例。

比如，ID从0到10000的用户会保存到实例R0，ID从10001到 20000的用户会保存到R1，以此类推。

这种方式是可行的，并且在实际中使用，不足就是要有一个区间范围到实例的映射表。这个表要被管理，同时还需要各 种对象的映射表，通常对Redis来说并非是好的方法。

哈希分区

另外一种分区方法是hash分区。这对任何key都适用，也无需是object_name:<id>这种形式，像下面描述的一样简单：</id>

- 用一个hash函数将key转换为一个数字，比如使用crc32 hash函数。对key foobar执行crc32(foobar)会输出类似93024922的整数。
- 对这个整数取模，将其转化为0-3之间的数字，就可以将这个整数映射到4个Redis实例中的一个了。93024922 % 4 = 2，就是说key foobar应该被存到R2实例中。注意：取模操作是取除的余数，通常在多种编程语言中用%操作符实现。

Java 使用 Redis

安装

开始在 Java 中使用 Redis 前， 我们需要确保已经安装了 redis 服务及 Java redis 驱动，且你的机器上能正常使用 Java。 Java的安装配置可以参考我们的 Java开发环境配置 接下来让我们安装 Java redis 驱动：

- 首先你需要下载驱动包，下载 **jedis.jar**，确保下载最新驱动包。
- 在你的classpath中包含该驱动包。

连接到 redis 服务

```
import redis.clients.jedis.Jedis;
public class RedisJava {
   public static void main(String[] args) {
      //连接本地的 Redis 服务
      Jedis jedis = new Jedis("localhost");
      System.out.println("Connection to server sucessfully");
      //查看服务是否运行
      System.out.println("Server is running: "+jedis.ping());
 }
}
```

编译以上 Java 程序，确保驱动包的路径是正确的。

```
$javac RedisJava.java
$java RedisJava
Connection to server sucessfully
Server is running: PONG

Redis Java String Example
```

Redis Java String(字符串) 实例

```
import redis.clients.jedis.Jedis;
public class RedisStringJava {
   public static void main(String[] args) {
      //连接本地的 Redis 服务
      Jedis jedis = new Jedis("localhost");
      System.out.println("Connection to server sucessfully");
      //设置 redis 字符串数据
      jedis.set("w3ckey", "Redis tutorial");
     // 获取存储的数据并输出
     System.out.println("Stored string in redis:: "+ jedis.get("w3c
 }
}
```

编译以上程序。

```
$javac RedisStringJava.java
$java RedisStringJava
Connection to server sucessfully
Stored string in redis:: Redis tutorial
```

Redis Java List(列表) 实例

```
import redis.clients.jedis.Jedis;
public class RedisListJava {
   public static void main(String[] args) {
      //连接本地的 Redis 服务
      Jedis jedis = new Jedis("localhost");
      System.out.println("Connection to server sucessfully");
      //存储数据到列表中
      jedis.lpush("tutorial-list", "Redis");
      jedis.lpush("tutorial-list", "Mongodb");
      jedis.lpush("tutorial-list", "Mysql");
     // 获取存储的数据并输出
     List<String> list = jedis.lrange("tutorial-list", 0 ,5);
     for(int i=0; i<list.size(); i++) {
       System.out.println("Stored string in redis:: "+list.get(i));
     }
 }
}
```

编译以上程序。

```
$javac RedisListJava.java
$java RedisListJava
Connection to server sucessfully
Stored string in redis:: Redis
Stored string in redis:: Mongodb
Stored string in redis:: Mysql
```

Redis Java Keys 实例

```
import redis.clients.jedis.Jedis;
public class RedisKeyJava {
   public static void main(String[] args) {
      //连接本地的 Redis 服务
      Jedis jedis = new Jedis("localhost");
      System.out.println("Connection to server sucessfully");

     // 获取数据并输出
     List<String> list = jedis.keys("*");
     for(int i=0; i<list.size(); i++) {
       System.out.println("List of stored keys:: "+list.get(i));
     }
   }
}
```

编译以上程序。

```
$javac RedisKeyJava.java
$java RedisKeyJava
Connection to server sucessfully
List of stored keys:: tutorial-name
List of stored keys:: tutorial-list
```

PHP 使用 Redis

安装

开始在 PHP 中使用 Redis 前， 我们需要确保已经安装了 redis 服务及 PHP redis 驱动，且你的机器上能正常使用 PHP。 接下来让我们安装 PHP redis 驱动：下载地址为:**https://github.com/nicolasff/phpredis**。

PHP安装redis扩展

```
/usr/local/php/bin/phpize              #php安装后的路径

./configure --with-php-config=/usr/local/php/bin/php-config

make && make install
```

修改php.ini文件

```
vi /usr/local/php/lib/php.ini
```

增加如下内容:

```
extension_dir = "/usr/local/php/lib/php/extensions/no-debug-zts-20(

extension=redis.so
```

安装完成后重启php-fpm 或 apache。查看phpinfo信息，就能看到redis扩展。

Phar based on pear/PHP_Archive, original concept by Davey Shafik.
Phar fully realized by Gregory Beaver and Marcus Boerger.
Portions of tar implementation Copyright (c) 2003-2009 Tim Kientzle.

Directive	Local Value	Master Value
phar.cache_list	*no value*	*no value*
phar.readonly	On	On
phar.require_hash	On	On

posix

Revision	$Id: 1dfa9997ed76804e53c91e0ce862f3707617b6ed $

redis

Redis Support	enabled
Redis Version	2.2.4

redis 第 1 条，共 6 条

连接到 redis 服务

```
<?php
    //连接本地的 Redis 服务
   $redis = new Redis();
   $redis->connect('127.0.0.1', 6379);
   echo "Connection to server sucessfully";
        //查看服务是否运行
   echo "Server is running: "+ $redis->ping();
?>
```

执行脚本，输出结果为：

```
Connection to server sucessfully
Server is running: PONG
```

Redis Java String(字符串) 实例

```
<?php
   //连接本地的 Redis 服务
   $redis = new Redis();
   $redis->connect('127.0.0.1', 6379);
   echo "Connection to server sucessfully";
   //设置 redis 字符串数据
   $redis->set("tutorial-name", "Redis tutorial");
   // 获取存储的数据并输出
   echo "Stored string in redis:: " + jedis.get("tutorial-name");
?>
```

执行脚本，输出结果为：

```
Connection to server sucessfully
Stored string in redis:: Redis tutorial
```

Redis Java List(列表) 实例

```
<?php
   //连接本地的 Redis 服务
   $redis = new Redis();
   $redis->connect('127.0.0.1', 6379);
   echo "Connection to server sucessfully";
   //存储数据到列表中
   $redis->lpush("tutorial-list", "Redis");
   $redis->lpush("tutorial-list", "Mongodb");
   $redis->lpush("tutorial-list", "Mysql");
   // 获取存储的数据并输出
   $arList = $redis->lrange("tutorial-list", 0 ,5);
   echo "Stored string in redis:: "
   print_r($arList);
?>
```

执行脚本，输出结果为：

```
Connection to server sucessfully
Stored string in redis::
Redis
Mongodb
Mysql
```

Redis Java Keys 实例

```
<?php
   //连接本地的 Redis 服务
   $redis = new Redis();
   $redis->connect('127.0.0.1', 6379);
   echo "Connection to server sucessfully";
   // 获取数据并输出
   $arList = $redis->keys("*");
   echo "Stored keys in redis:: "
   print_r($arList);
?>
```

执行脚本，输出结果为：

```
Connection to server sucessfully
Stored string in redis::
tutorial-name
tutorial-list
```

DEL

DEL key [key ...]

删除给定的一个或多个 `key` 。

不存在的 `key` 会被忽略。

可用版本：

>= 1.0.0

时间复杂度：

O(N)， `N` 为被删除的 `key` 的数量。删除单个字符串类型的 `key` ，时间复杂度为O(1)。删除单个列表、集合、有序集合或哈希表类型的 `key` ，时间复杂度为O(M)， `M` 为以上数据结构内的元素数量。

返回值：

被删除 `key` 的数量。

```
#  删除单个 key

redis> SET name huangz
OK

redis> DEL name
(integer) 1

# 删除一个不存在的 key

redis> EXISTS phone
(integer) 0

redis> DEL phone # 失败，没有 key 被删除
(integer) 0

# 同时删除多个 key

redis> SET name "redis"
OK

redis> SET type "key-value store"
OK

redis> SET website "redis.com"
OK

redis> DEL name type website
(integer) 3
```

DUMP

DUMP key

序列化给定 `key` ，并返回被序列化的值，使用 *RESTORE* 命令可以将这个值反序列化为 Redis 键。

序列化生成的值有以下几个特点：

- 它带有 64 位的校验和，用于检测错误， *RESTORE* 在进行反序列化之前会先检查校验和。
- 值的编码格式和 RDB 文件保持一致。
- RDB 版本会被编码在序列化值当中，如果因为 Redis 的版本不同造成 RDB 格式不兼容，那么 Redis 会拒绝对这个值进行反序列化操作。

序列化的值不包括任何生存时间信息。

可用版本：

>= 2.6.0

时间复杂度：

查找给定键的复杂度为 O(1) ，对键进行序列化的复杂度为 O(N*M) ，其中 N 是构成 `key` 的 Redis 对象的数量，而 M 则是这些对象的平均大小。如果序列化的对象是比较小的字符串，那么复杂度为 O(1) 。

返回值：

如果 `key` 不存在，那么返回 `nil` 。否则，返回序列化之后的值。

```
redis> SET greeting "hello, dumping world!"
OK

redis> DUMP greeting
"\x00\x15hello, dumping world!\x06\x00E\xa0Z\x82\xd8r\xc1\xde"

redis> DUMP not-exists-key
(nil)
```

Redis 命令参考

Key（键）

EXISTS

EXISTS key

检查给定 `key` 是否存在。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

若 `key` 存在，返回 `1` ，否则返回 `0` 。

```
redis> SET db "redis"
OK

redis> EXISTS db
(integer) 1

redis> DEL db
(integer) 1

redis> EXISTS db
(integer) 0
```

EXPIRE

EXPIRE key seconds

为给定 `key` 设置生存时间，当 `key` 过期时(生存时间为 `0`)，它会被自动删除。

在 Redis 中，带有生存时间的 `key` 被称为『易失的』(volatile)。

生存时间可以通过使用 *DEL* 命令来删除整个 `key` 来移除，或者被 *SET* 和 *GETSET* 命令覆写(overwrite)，这意味着，如果一个命令只是修改(alter)一个带生存时间的 `key` 的值而不是用一个新的 `key` 值来代替(replace)它的话，那么生存时间不会被改变。

比如说，对一个 `key` 执行 *INCR* 命令，对一个列表进行 *LPUSH* 命令，或者对一个哈希表执行 *HSET* 命令，这类操作都不会修改 `key` 本身的生存时间。

另一方面，如果使用 *RENAME* 对一个 `key` 进行改名，那么改名后的 `key` 的生存时间和改名前一样。

RENAME 命令的另一种可能是，尝试将一个带生存时间的 `key` 改名成另一个带生存时间的 `another_key` ，这时旧的 `another_key` (以及它的生存时间)会被删除，然后旧的 `key` 会改名为 `another_key` ，因此，新的 `another_key` 的生存时间也和原本的 `key` 一样。

使用 *PERSIST* 命令可以在不删除 `key` 的情况下，移除 `key` 的生存时间，让 `key` 重新成为一个『持久的』(persistent) `key` 。

更新生存时间

可以对一个已经带有生存时间的 `key` 执行 *EXPIRE* 命令，新指定的生存时间会取代旧的生存时间。

过期时间的精确度

在 Redis 2.4 版本中，过期时间的延迟在 1 秒钟之内 —— 也即是，就算 `key` 已经过期，但它还是可能在过期之后一秒钟之内被访问到，而在新的 Redis 2.6 版本中，延迟被降低到 1 毫秒之内。

Redis 2.1.3 之前的不同之处

在 Redis 2.1.3 之前的版本中，修改一个带有生存时间的 `key` 会导致整个 `key` 被删除，这一行为是受当时复制(replication)层的限制而作出的，现在这一限制已经被修复。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

设置成功返回 `1` 。当 `key` 不存在或者不能为 `key` 设置生存时间时(比如在低于 2.1.3 版本的 Redis 中你尝试更新 `key` 的生存时间)，返回 `0` 。

```
redis> SET cache_page "www.google.com"
OK

redis> EXPIRE cache_page 30  # 设置过期时间为 30 秒
(integer) 1

redis> TTL cache_page    # 查看剩余生存时间
(integer) 23

redis> EXPIRE cache_page 30000   # 更新过期时间
(integer) 1

redis> TTL cache_page
(integer) 29996
```

模式：导航会话

假设你有一项 web 服务，打算根据用户最近访问的 N 个页面来进行物品推荐，并且假设用户停止阅览超过 60 秒，那么就清空阅览记录(为了减少物品推荐的计算量，并且保持推荐物品的新鲜度)。

这些最近访问的页面记录，我们称之为『导航会话』(Navigation session)，可以用 *INCR* 和 *RPUSH* 命令在 Redis 中实现它：每当用户阅览一个网页的时候，执行以下代码：

```
MULTI
    RPUSH pagewviews.user:<userid> http://.....
    EXPIRE pagewviews.user:<userid> 60
EXEC
```

如果用户停止阅览超过 60 秒，那么它的导航会话就会被清空，当用户重新开始阅览的时候，系统又会重新记录导航会话，继续进行物品推荐。

EXPIREAT

EXPIREAT key timestamp

EXPIREAT 的作用和 *EXPIRE* 类似，都用于为 `key` 设置生存时间。

不同在于 EXPIREAT 命令接受的时间参数是 UNIX 时间戳(unix timestamp)。

可用版本：

>= 1.2.0

时间复杂度：

O(1)

返回值：

如果生存时间设置成功，返回 `1` 。当 `key` 不存在或没办法设置生存时间，返回 `0` 。

```
redis> SET cache www.google.com
OK

redis> EXPIREAT cache 1355292000     # 这个 key 将在 2012.12.12 过期
(integer) 1

redis> TTL cache
(integer) 45081860
```

KEYS

KEYS pattern

查找所有符合给定模式 `pattern` 的 `key` 。

`KEYS *` 匹配数据库中所有 `key` 。 `KEYS h?llo` 匹配 `hello` ， `hallo` 和 `hxllo` 等。 `KEYS h*llo` 匹配 `hllo` 和 `heeeeello` 等。 `KEYS h[ae]llo` 匹配 `hello` 和 `hallo` ，但不匹配 `hillo` 。

特殊符号用 `\` 隔开

Warning

KEYS 的速度非常快，但在一个大的数据库中使用它仍然可能造成性能问题，如果你需要从一个数据集中查找特定的 `key` ，你最好还是用 Redis 的集合结构(set)来代替。

可用版本：

>= 1.0.0

时间复杂度：

O(N)， `N` 为数据库中 `key` 的数量。

返回值：

符合给定模式的 `key` 列表。

```
redis> MSET one 1 two 2 three 3 four 4  # 一次设置 4 个 key
OK

redis> KEYS *o*
1) "four"
2) "two"
3) "one"

redis> KEYS t??
1) "two"

redis> KEYS t[w]*
1) "two"

redis> KEYS *  # 匹配数据库内所有 key
1) "four"
2) "three"
3) "two"
4) "one"
```

MIGRATE

MIGRATE host port key destination-db timeout [COPY] [REPLACE]

将 `key` 原子性地从当前实例传送到目标实例的指定数据库上，一旦传送成功， `key` 保证会出现在目标实例上，而当前实例上的 `key` 会被删除。

这个命令是一个原子操作，它在执行的时候会阻塞进行迁移的两个实例，直到以下任意结果发生：迁移成功，迁移失败，等到超时。

命令的内部实现是这样的：它在当前实例对给定 `key` 执行 *DUMP* 命令，将它序列化，然后传送到目标实例，目标实例再使用 *RESTORE* 对数据进行反序列化，并将反序列化所得的数据添加到数据库中；当前实例就像目标实例的客户端那样，只要看到 *RESTORE* 命令返回 `OK` ，它就会调用 *DEL* 删除自己数据库上的 `key` 。

`timeout` 参数以毫秒为格式，指定当前实例和目标实例进行沟通的最大间隔时间。这说明操作并不一定要在 `timeout` 毫秒内完成，只是说数据传送的时间不能超过这个 `timeout` 数。

MIGRATE 命令需要在给定的时间规定内完成 IO 操作。如果在传送数据时发生 IO 错误，或者达到了超时时间，那么命令会停止执行，并返回一个特殊的错误： `IOERR` 。

当 `IOERR` 出现时，有以下两种可能：

- `key` 可能存在于两个实例
- `key` 可能只存在于当前实例

唯一不可能发生的情况就是丢失 `key` ，因此，如果一个客户端执行 MIGRATE 命令，并且不幸遇上 `IOERR` 错误，那么这个客户端唯一要做的就是检查自己数据库上的 `key` 是否已经被正确地删除。

如果有其他错误发生，那么 MIGRATE 保证 `key` 只会出现在当前实例中。（当然，目标实例的给定数据库上可能有和 `key` 同名的键，不过这和 MIGRATE 命令没有关系）。

可选项：

- `COPY` ：不移除源实例上的 `key` 。
- `REPLACE` ：替换目标实例上已存在的 `key` 。

可用版本：

>= 2.6.0

时间复杂度：

这个命令在源实例上实际执行 *DUMP* 命令和 *DEL* 命令，在目标实例执行 *RESTORE* 命令，查看以上命令的文档可以看到详细的复杂度说明。 `key` 数据在两个实例之间传输的复杂度为 O(N) 。

返回值：

迁移成功时返回 `OK` ，否则返回相应的错误。

示例

先启动两个 Redis 实例，一个使用默认的 6379 端口，一个使用 7777 端口。

```
$ ./redis-server &
[1] 3557

...

$ ./redis-server --port 7777 &
[2] 3560

...
```

然后用客户端连上 6379 端口的实例，设置一个键，然后将它迁移到 7777 端口的实例上：

```
$ ./redis-cli

redis 127.0.0.1:6379> flushdb
OK

redis 127.0.0.1:6379> SET greeting "Hello from 6379 instance"
OK

redis 127.0.0.1:6379> MIGRATE 127.0.0.1 7777 greeting 0 1000
OK

redis 127.0.0.1:6379> EXISTS greeting                                    # 
(integer) 0
```

使用另一个客户端，查看 7777 端口上的实例：

```
$ ./redis-cli -p 7777

redis 127.0.0.1:7777> GET greeting
"Hello from 6379 instance"
```

MOVE

MOVE key db

将当前数据库的 `key` 移动到给定的数据库 `db` 当中。

如果当前数据库(源数据库)和给定数据库(目标数据库)有相同名字的给定 `key` ，或者 `key` 不存在于当前数据库，那么 `MOVE` 没有任何效果。

因此，也可以利用这一特性，将 MOVE 当作锁(locking)原语(primitive)。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

移动成功返回 `1` ，失败则返回 `0` 。

```
# key 存在于当前数据库

redis> SELECT 0                             # redis默认使用数据库 0,
OK

redis> SET song "secret base - Zone"
OK

redis> MOVE song 1                          # 将 song 移动到数据库 1
(integer) 1

redis> EXISTS song                          # song 已经被移走
(integer) 0

redis> SELECT 1                             # 使用数据库 1
OK

redis:1> EXISTS song                        # 证实 song 被移到了数据库
(integer) 1

# 当 key 不存在的时候

redis:1> EXISTS fake_key
(integer) 0

redis:1> MOVE fake_key 0                    # 试图从数据库 1 移动一个不
(integer) 0
```

```
redis:1> select 0                            # 使用数据库0
OK

redis> EXISTS fake_key                       # 证实 fake_key 不存在
(integer) 0

# 当源数据库和目标数据库有相同的 key 时

redis> SELECT 0                              # 使用数据库0
OK
redis> SET favorite_fruit "banana"
OK

redis> SELECT 1                              # 使用数据库1
OK
redis:1> SET favorite_fruit "apple"
OK

redis:1> SELECT 0                            # 使用数据库0，并试图将 fa
OK

redis> MOVE favorite_fruit 1                 # 因为两个数据库有相同的 k
(integer) 0

redis> GET favorite_fruit                    # 数据库 0 的 favorite_f
"banana"

redis> SELECT 1
OK

redis:1> GET favorite_fruit                  # 数据库 1 的 favorite_f
"apple"
```

OBJECT

OBJECT subcommand [arguments [arguments]]

OBJECT 命令允许从内部察看给定 `key` 的 Redis 对象。

它通常用在除错(debugging)或者了解为了节省空间而对 `key` 使用特殊编码的情况。当将Redis用作缓存程序时，你也可以通过 OBJECT 命令中的信息，决定 `key` 的驱逐策略(eviction policies)。

OBJECT 命令有多个子命令：

- `OBJECT REFCOUNT <key>` 返回给定 `key` 引用所储存的值的次数。此命令主要用于除错。
- `OBJECT ENCODING <key>` 返回给定 `key` 锁储存的值所使用的内部表示(representation)。
- `OBJECT IDLETIME <key>` 返回给定 `key` 自储存以来的空闲时间(idle， 没有被读取也没有被写入)，以秒为单位。对象可以以多种方式编码：
- 字符串可以被编码为 `raw` (一般字符串)或 `int` (为了节约内存，Redis 会将字符串表示的 64 位有符号整数编码为整数来进行储存）。
- 列表可以被编码为 `ziplist` 或 `linkedlist` 。 `ziplist` 是为节约大小较小的列表空间而作的特殊表示。
- 集合可以被编码为 `intset` 或者 `hashtable` 。 `intset` 是只储存数字的小集合的特殊表示。
- 哈希表可以编码为 `zipmap` 或者 `hashtable` 。 `zipmap` 是小哈希表的特殊表示。
- 有序集合可以被编码为 `ziplist` 或者 `skiplist` 格式。 `ziplist` 用于表示小的有序集合，而 `skiplist` 则用于表示任何大小的有序集合。假如你做了什么让 Redis 没办法再使用节省空间的编码时(比如将一个只有 1 个元素的集合扩展为一个有 100 万个元素的集合)，特殊编码类型(specially encoded types)会自动转换成通用类型(general type)。

可用版本：

>= 2.2.3

时间复杂度：

O(1)

返回值：

`REFCOUNT` 和 `IDLETIME` 返回数字。 `ENCODING` 返回相应的编码类型。

```
redis> SET game "COD"           # 设置一个字符串
OK

redis> OBJECT REFCOUNT game     # 只有一个引用
(integer) 1

redis> OBJECT IDLETIME game     # 等待一阵。。。然后查看空闲时间
(integer) 90

redis> GET game                 # 提取game， 让它处于活跃(active)状态
"COD"

redis> OBJECT IDLETIME game     # 不再处于空闲状态
(integer) 0

redis> OBJECT ENCODING game     # 字符串的编码方式
"raw"

redis> SET big-number 23102930128301091820391092019203810281029831(
OK

redis> OBJECT ENCODING big-number
"raw"

redis> SET small-number 12345  # 而短的数字则会被编码为整数
OK

redis> OBJECT ENCODING small-number
"int"
```

PERSIST

PERSIST key

移除给定 `key` 的生存时间，将这个 `key` 从『易失的』(带生存时间 `key`)转换成『持久的』(一个不带生存时间、永不过期的 `key`)。

可用版本：

>= 2.2.0

时间复杂度：

O(1)

返回值：

当生存时间移除成功时，返回 `1` .如果 `key` 不存在或 `key` 没有设置生存时间，返回 `0` 。

```
redis> SET mykey "Hello"
OK

redis> EXPIRE mykey 10  # 为 key 设置生存时间
(integer) 1

redis> TTL mykey
(integer) 10

redis> PERSIST mykey    # 移除 key 的生存时间
(integer) 1

redis> TTL mykey
(integer) -1
```

PEXPIRE

PEXPIRE key milliseconds

这个命令和 *EXPIRE* 命令的作用类似，但是它以毫秒为单位设置 `key` 的生存时间，而不像 *EXPIRE* 命令那样，以秒为单位。

可用版本：

>= 2.6.0

时间复杂度：

O(1)

返回值：

设置成功，返回 `1``key` 不存在或设置失败，返回 `0`

```
redis> SET mykey "Hello"
OK

redis> PEXPIRE mykey 1500
(integer) 1

redis> TTL mykey    # TTL 的返回值以秒为单位
(integer) 2

redis> PTTL mykey   # PTTL 可以给出准确的毫秒数
(integer) 1499
```

PEXPIREAT

PEXPIREAT key milliseconds-timestamp

这个命令和 *EXPIREAT* 命令类似，但它以毫秒为单位设置 `key` 的过期 unix 时间戳，而不是像 *EXPIREAT* 那样，以秒为单位。

可用版本：

>= 2.6.0

时间复杂度：

O(1)

返回值：

如果生存时间设置成功，返回 `1` 。当 `key` 不存在或没办法设置生存时间时，返回 `0` 。(查看 *EXPIRE* 命令获取更多信息)

```
redis> SET mykey "Hello"
OK

redis> PEXPIREAT mykey 1555555555005
(integer) 1

redis> TTL mykey           # TTL 返回秒
(integer) 223157079

redis> PTTL mykey          # PTTL 返回毫秒
(integer) 223157079318
```

PTTL

PTTL key

这个命令类似于 *TTL* 命令，但它以毫秒为单位返回 `key` 的剩余生存时间，而不是像 *TTL* 命令那样，以秒为单位。

可用版本：

>= 2.6.0

复杂度：

O(1)

返回值：

当 `key` 不存在时，返回 `-2` 。当 `key` 存在但没有设置剩余生存时间时，返回 `-1` 。否则，以毫秒为单位，返回 `key` 的剩余生存时间。

Note

在 Redis 2.8 以前，当 `key` 不存在，或者 `key` 没有设置剩余生存时间时，命令都返回 `-1` 。

```
# 不存在的 key

redis> FLUSHDB
OK

redis> PTTL key
(integer) -2

# key 存在，但没有设置剩余生存时间

redis> SET key value
OK

redis> PTTL key
(integer) -1

# 有剩余生存时间的 key

redis> PEXPIRE key 10086
(integer) 1

redis> PTTL key
(integer) 6179
```

RANDOMKEY

RANDOMKEY

从当前数据库中随机返回(不删除)一个 `key` 。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

当数据库不为空时，返回一个 `key` 。当数据库为空时，返回 `nil` 。

```
# 数据库不为空

redis> MSET fruit "apple" drink "beer" food "cookies"   # 设置多个 k
OK

redis> RANDOMKEY
"fruit"

redis> RANDOMKEY
"food"

redis> KEYS *    # 查看数据库内所有key，证明 RANDOMKEY 并不删除 key
1) "food"
2) "drink"
3) "fruit"

# 数据库为空

redis> FLUSHDB  # 删除当前数据库所有 key
OK

redis> RANDOMKEY
(nil)
```

RENAME

RENAME key newkey

将 `key` 改名为 `newkey` 。

当 `key` 和 `newkey` 相同，或者 `key` 不存在时，返回一个错误。

当 `newkey` 已经存在时， RENAME 命令将覆盖旧值。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

改名成功时提示 `OK` ，失败时候返回一个错误。

```
# key 存在且 newkey 不存在

redis> SET message "hello world"
OK

redis> RENAME message greeting
OK

redis> EXISTS message                    # message 不复存在
(integer) 0

redis> EXISTS greeting                   # greeting 取而代之
(integer) 1

# 当 key 不存在时，返回错误

redis> RENAME fake_key never_exists
(error) ERR no such key

# newkey 已存在时， RENAME 会覆盖旧 newkey

redis> SET pc "lenovo"
OK

redis> SET personal_computer "dell"
OK

redis> RENAME pc personal_computer
OK

redis> GET pc
(nil)

redis:1> GET personal_computer           # 原来的值 dell 被覆盖了
"lenovo"
```

RENAMENX

RENAMENX key newkey

当且仅当 `newkey` 不存在时，将 `key` 改名为 `newkey` 。

当 `key` 不存在时，返回一个错误。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

修改成功时，返回 `1` 。如果 `newkey` 已经存在，返回 `0` 。

```
# newkey 不存在，改名成功

redis> SET player "MPlyaer"
OK

redis> EXISTS best_player
(integer) 0

redis> RENAMENX player best_player
(integer) 1

# newkey存在时，失败

redis> SET animal "bear"
OK

redis> SET favorite_animal "butterfly"
OK

redis> RENAMENX animal favorite_animal
(integer) 0

redis> get animal
"bear"

redis> get favorite_animal
"butterfly"
```

RESTORE

RESTORE key ttl serialized-value [REPLACE]

反序列化给定的序列化值，并将它和给定的 `key` 关联。

参数 `ttl` 以毫秒为单位为 `key` 设置生存时间；如果 `ttl` 为 `0` ，那么不设置生存时间。

RESTORE 在执行反序列化之前会先对序列化值的 RDB 版本和数据校验和进行检查，如果 RDB 版本不相同或者数据不完整的话，那么 RESTORE 会拒绝进行反序列化，并返回一个错误。

如果键 `key` 已经存在， 并且给定了 `REPLACE` 选项， 那么使用反序列化得出的值来代替键 `key` 原有的值； 相反地， 如果键 `key` 已经存在， 但是没有给定 `REPLACE` 选项， 那么命令返回一个错误。

更多信息可以参考 *DUMP* 命令。

可用版本：

>= 2.6.0

时间复杂度：

查找给定键的复杂度为 O(1) ，对键进行反序列化的复杂度为 O(N*M*) ，其中 *N* 是构成 *key* 的 *Redis* 对象的数量，而 *M* 则是这些对象的平均大小。有序集合*(sorted set)*的反序列化复杂度为 *O(N*M*log(N))* ，因为有序集合每次插入的复杂度为 O(log(N)) 。如果反序列化的对象是比较小的字符串，那么复杂度为 O(1) 。

返回值：

如果反序列化成功那么返回 `OK` ，否则返回一个错误。

```
# 创建一个键，作为 DUMP 命令的输入

redis> SET greeting "hello, dumping world!"
OK

redis> DUMP greeting
"\x00\x15hello, dumping world!\x06\x00E\xa0Z\x82\xd8r\xc1\xde"

# 将序列化数据 RESTORE 到另一个键上面

redis> RESTORE greeting-again 0 "\x00\x15hello, dumping world!\x06\
OK

redis> GET greeting-again
"hello, dumping world!"

# 在没有给定 REPLACE 选项的情况下，再次尝试反序列化到同一个键，失败

redis> RESTORE greeting-again 0 "\x00\x15hello, dumping world!\x06\
(error) ERR Target key name is busy.

# 给定 REPLACE 选项，对同一个键进行反序列化成功

redis> RESTORE greeting-again 0 "\x00\x15hello, dumping world!\x06\
OK

# 尝试使用无效的值进行反序列化，出错

redis> RESTORE fake-message 0 "hello moto moto blah blah"
(error) ERR DUMP payload version or checksum are wrong
```

SORT

SORT key [BY pattern] [LIMIT offset count] [GET pattern [GET pattern ...]] [ASC | DESC] [ALPHA] [STORE destination]

返回或保存给定列表、集合、有序集合 `key` 中经过排序的元素。

排序默认以数字作为对象，值被解释为双精度浮点数，然后进行比较。

一般 SORT 用法

最简单的 SORT 使用方法是 `SORT key` 和 `SORT key DESC` ：

- `SORT key` 返回键值从小到大排序的结果。
- `SORT key DESC` 返回键值从大到小排序的结果。

假设 `today_cost` 列表保存了今日的开销金额，那么可以用 SORT 命令对它进行排序：

```
# 开销金额列表

redis> LPUSH today_cost 30 1.5 10 8
(integer) 4

# 排序

redis> SORT today_cost
1) "1.5"
2) "8"
3) "10"
4) "30"

# 逆序排序

redis 127.0.0.1:6379> SORT today_cost DESC
1) "30"
2) "10"
3) "8"
4) "1.5"
```

使用 ALPHA 修饰符对字符串进行排序

因为 SORT 命令默认排序对象为数字，当需要对字符串进行排序时，需要显式地在 SORT 命令之后添加 `ALPHA` 修饰符：

```
# 网址

redis> LPUSH website "www.reddit.com"
(integer) 1

redis> LPUSH website "www.slashdot.com"
(integer) 2

redis> LPUSH website "www.infoq.com"
(integer) 3

# 默认（按数字）排序

redis> SORT website
1) "www.infoq.com"
2) "www.slashdot.com"
3) "www.reddit.com"

# 按字符排序

redis> SORT website ALPHA
1) "www.infoq.com"
2) "www.reddit.com"
3) "www.slashdot.com"
```

如果系统正确地设置了 `LC_COLLATE` 环境变量的话，Redis能识别 `UTF-8` 编码。

使用 LIMIT 修饰符限制返回结果

排序之后返回元素的数量可以通过 `LIMIT` 修饰符进行限制， 修饰符接受 `offset` 和 `count` 两个参数：

- `offset` 指定要跳过的元素数量。
- `count` 指定跳过 `offset` 个指定的元素之后，要返回多少个对象。

以下例子返回排序结果的前 5 个对象(`offset` 为 `0` 表示没有元素被跳过)。

```
# 添加测试数据，列表值为 1 指 10

redis 127.0.0.1:6379> RPUSH rank 1 3 5 7 9
(integer) 5

redis 127.0.0.1:6379> RPUSH rank 2 4 6 8 10
(integer) 10

# 返回列表中最小的 5 个值

redis 127.0.0.1:6379> SORT rank LIMIT 0 5
1) "1"
2) "2"
3) "3"
4) "4"
5) "5"
```

可以组合使用多个修饰符。以下例子返回从大到小排序的前 5 个对象。

```
redis 127.0.0.1:6379> SORT rank LIMIT 0 5 DESC
1) "10"
2) "9"
3) "8"
4) "7"
5) "6"
```

使用外部 **key** 进行排序

可以使用外部 `key` 的数据作为权重，代替默认的直接对比键值的方式来进行排序。

假设现在有用户数据如下：

uid	user*name*{uid}	user*level*{uid}
1	admin	9999
2	jack	10
3	peter	25
4	mary	70

以下代码将数据输入到 Redis 中：

```
# admin

redis 127.0.0.1:6379> LPUSH uid 1
(integer) 1

redis 127.0.0.1:6379> SET user_name_1 admin
OK

redis 127.0.0.1:6379> SET user_level_1 9999
OK

# jack

redis 127.0.0.1:6379> LPUSH uid 2
(integer) 2

redis 127.0.0.1:6379> SET user_name_2 jack
OK

redis 127.0.0.1:6379> SET user_level_2 10
OK

# peter

redis 127.0.0.1:6379> LPUSH uid 3
(integer) 3

redis 127.0.0.1:6379> SET user_name_3 peter
OK

redis 127.0.0.1:6379> SET user_level_3 25
OK

# mary

redis 127.0.0.1:6379> LPUSH uid 4
(integer) 4

redis 127.0.0.1:6379> SET user_name_4 mary
OK

redis 127.0.0.1:6379> SET user_level_4 70
OK
```

BY 选项

默认情况下， `SORT uid` 直接按 `uid` 中的值排序：

```
redis 127.0.0.1:6379> SORT uid
1) "1"      # admin
2) "2"      # jack
3) "3"      # peter
4) "4"      # mary
```

通过使用 `BY` 选项，可以让 `uid` 按其他键的元素来排序。

比如说， 以下代码让 `uid` 键按照 `user_level_{uid}` 的大小来排序：

```
redis 127.0.0.1:6379> SORT uid BY user_level_*
1) "2"      # jack , level = 10
2) "3"      # peter, level = 25
3) "4"      # mary, level = 70
4) "1"      # admin, level = 9999
```

`user_level_*` 是一个占位符， 它先取出 `uid` 中的值， 然后再用这个值来查找相应的键。

比如在对 `uid` 列表进行排序时， 程序就会先取出 `uid` 的值 `1` 、 `2` 、 `3` 、 `4` ， 然后使用 `user_level_1` 、 `user_level_2` 、 `user_level_3` 和 `user_level_4` 的值作为排序 `uid` 的权重。

GET 选项

使用 `GET` 选项， 可以根据排序的结果来取出相应的键值。

比如说， 以下代码先排序 `uid` ， 再取出键 `user_name_{uid}` 的值：

```
redis 127.0.0.1:6379> SORT uid GET user_name_*
1) "admin"
2) "jack"
3) "peter"
4) "mary"
```

组合使用 BY 和 GET

通过组合使用 `BY` 和 `GET` ， 可以让排序结果以更直观的方式显示出来。

比如说， 以下代码先按 `user_level_{uid}` 来排序 `uid` 列表， 再取出相应的 `user_name_{uid}` 的值：

```
redis 127.0.0.1:6379> SORT uid BY user_level_* GET user_name_*
1) "jack"        # level = 10
2) "peter"       # level = 25
3) "mary"        # level = 70
4) "admin"       # level = 9999
```

现在的排序结果要比只使用 `SORT uid BY user_level_*` 要直观得多。

获取多个外部键

可以同时使用多个 `GET` 选项，获取多个外部键的值。

以下代码就按 `uid` 分别获取 `user_level_{uid}` 和 `user_name_{uid}` ：

```
redis 127.0.0.1:6379> SORT uid GET user_level_* GET user_name_*
1) "9999"        # level
2) "admin"       # name
3) "10"
4) "jack"
5) "25"
6) "peter"
7) "70"
8) "mary"
```

`GET` 有一个额外的参数规则，那就是 —— 可以用 `#` 获取被排序键的值。

以下代码就将 `uid` 的值、及其相应的 `user_level_*` 和 `user_name_*` 都返回为结果：

```
redis 127.0.0.1:6379> SORT uid GET # GET user_level_* GET user_name
1) "1"           # uid
2) "9999"        # level
3) "admin"       # name
4) "2"
5) "10"
6) "jack"
7) "3"
8) "25"
9) "peter"
10) "4"
11) "70"
12) "mary"
```

获取外部键，但不进行排序

通过将一个不存在的键作为参数传给 `BY` 选项， 可以让 `SORT` 跳过排序操作，直接返回结果：

```
redis 127.0.0.1:6379> SORT uid BY not-exists-key
1) "4"
2) "3"
3) "2"
4) "1"
```

这种用法在单独使用时，没什么实际用处。

不过，通过将这种用法和 `GET` 选项配合， 就可以在不排序的情况下， 获取多个外部键， 相当于执行一个整合的获取操作（类似于 SQL 数据库的 `join` 关键字）。

以下代码演示了，如何在不引起排序的情况下，使用 `SORT` 、 `BY` 和 `GET` 获取多个外部键：

```
redis 127.0.0.1:6379> SORT uid BY not-exists-key GET # GET user_lev
1) "4"      # id
2) "70"     # level
3) "mary"   # name
4) "3"
5) "25"
6) "peter"
7) "2"
8) "10"
9) "jack"
10) "1"
11) "9999"
12) "admin"
```

将哈希表作为 GET 或 BY 的参数

除了可以将字符串键之外， 哈希表也可以作为 `GET` 或 `BY` 选项的参数来使用。

比如说，对于前面给出的用户信息表：

uid	user*name*{uid}	user*level*{uid}
1	admin	9999
2	jack	10
3	peter	25
4	mary	70

我们可以不将用户的名字和级别保存在 `user_name_{uid}` 和 `user_level_{uid}` 两个字符串键中， 而是用一个带有 `name` 域和 `level` 域的哈希表 `user_info_{uid}` 来保存用户的名字和级别信息：

```
redis 127.0.0.1:6379> HMSET user_info_1 name admin level 9999
OK

redis 127.0.0.1:6379> HMSET user_info_2 name jack level 10
OK

redis 127.0.0.1:6379> HMSET user_info_3 name peter level 25
OK

redis 127.0.0.1:6379> HMSET user_info_4 name mary level 70
OK
```

之后， `BY` 和 `GET` 选项都可以用 `key->field` 的格式来获取哈希表中的域的值， 其中 `key` 表示哈希表键， 而 `field` 则表示哈希表的域：

```
redis 127.0.0.1:6379> SORT uid BY user_info_*->level
1) "2"
2) "3"
3) "4"
4) "1"

redis 127.0.0.1:6379> SORT uid BY user_info_*->level GET user_info_
1) "jack"
2) "peter"
3) "mary"
4) "admin"
```

保存排序结果

默认情况下， SORT 操作只是简单地返回排序结果，并不进行任何保存操作。

通过给 `STORE` 选项指定一个 `key` 参数，可以将排序结果保存到给定的键上。

如果被指定的 `key` 已存在，那么原有的值将被排序结果覆盖。

```
# 测试数据

redis 127.0.0.1:6379> RPUSH numbers 1 3 5 7 9
(integer) 5

redis 127.0.0.1:6379> RPUSH numbers 2 4 6 8 10
(integer) 10

redis 127.0.0.1:6379> LRANGE numbers 0 -1
1) "1"
2) "3"
3) "5"
4) "7"
5) "9"
6) "2"
7) "4"
8) "6"
9) "8"
10) "10"

redis 127.0.0.1:6379> SORT numbers STORE sorted-numbers
(integer) 10

# 排序后的结果

redis 127.0.0.1:6379> LRANGE sorted-numbers 0 -1
1) "1"
2) "2"
3) "3"
4) "4"
5) "5"
6) "6"
7) "7"
8) "8"
9) "9"
10) "10"
```

可以通过将 SORT 命令的执行结果保存，并用 *EXPIRE* 为结果设置生存时间，以此来产生一个 SORT 操作的结果缓存。

这样就可以避免对 SORT 操作的频繁调用：只有当结果集过期时，才需要再调用一次 SORT 操作。

另外，为了正确实现这一用法，你可能需要加锁以避免多个客户端同时进行缓存重建(也就是多个客户端，同一时间进行 SORT 操作，并保存为结果集)，具体参见 *SETNX* 命令。

可用版本：

>= 1.0.0

时间复杂度：

O(N+M*log(M))， `N` 为要排序的列表或集合内的元素数量， `M` 为要返回的元素数量。如果只是使用 SORT 命令的 `GET` 选项获取数据而没有进行排序，时间复杂度 O(N)。

返回值：

没有使用 `STORE` 参数，返回列表形式的排序结果。使用 `STORE` 参数，返回排序结果的元素数量。

TTL

TTL key

以秒为单位，返回给定 `key` 的剩余生存时间(TTL, time to live)。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

当 `key` 不存在时，返回 `-2` 。当 `key` 存在但没有设置剩余生存时间时，返回 `-1` 。否则，以秒为单位，返回 `key` 的剩余生存时间。

Note

在 Redis 2.8 以前，当 `key` 不存在，或者 `key` 没有设置剩余生存时间时，命令都返回 `-1` 。

```
# 不存在的 key

redis> FLUSHDB
OK

redis> TTL key
(integer) -2

# key 存在，但没有设置剩余生存时间

redis> SET key value
OK

redis> TTL key
(integer) -1

# 有剩余生存时间的 key

redis> EXPIRE key 10086
(integer) 1

redis> TTL key
(integer) 10084
```

TYPE

TYPE key

返回 `key` 所储存的值的类型。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

`none` (key不存在) `string` (字符串) `list` (列表) `set` (集合) `zset` (有序集) `hash` (哈希表)

```
# 字符串

redis> SET weather "sunny"
OK

redis> TYPE weather
string

# 列表

redis> LPUSH book_list "programming in scala"
(integer) 1

redis> TYPE book_list
list

# 集合

redis> SADD pat "dog"
(integer) 1

redis> TYPE pat
set
```

SCAN

SCAN cursor [MATCH pattern] [COUNT count]

SCAN 命令及其相关的 *SSCAN* 命令、 *HSCAN* 命令和 *ZSCAN* 命令都用于增量地迭代（incrementally iterate）一集元素（a collection of elements）：

- *SCAN* 命令用于迭代当前数据库中的数据库键。
- *SSCAN* 命令用于迭代集合键中的元素。
- *HSCAN* 命令用于迭代哈希键中的键值对。
- *ZSCAN* 命令用于迭代有序集合中的元素（包括元素成员和元素分值）。

以上列出的四个命令都支持增量式迭代， 它们每次执行都只会返回少量元素， 所以这些命令可以用于生产环境， 而不会出现像 *KEYS* 命令、 *SMEMBERS* 命令带来的问题 —— 当 *KEYS* 命令被用于处理一个大的数据库时， 又或者 *SMEMBERS* 命令被用于处理一个大的集合键时， 它们可能会阻塞服务器达数秒之久。

不过， 增量式迭代命令也不是没有缺点的： 举个例子， 使用 *SMEMBERS* 命令可以返回集合键当前包含的所有元素， 但是对于 *SCAN* 这类增量式迭代命令来说， 因为在对键进行增量式迭代的过程中， 键可能会被修改， 所以增量式迭代命令只能对被返回的元素提供有限的保证 （offer limited guarantees about the returned elements）。

因为 *SCAN* 、 *SSCAN* 、 *HSCAN* 和 *ZSCAN* 四个命令的工作方式都非常相似， 所以这个文档会一并介绍这四个命令， 但是要记住：

- *SSCAN* 命令、 *HSCAN* 命令和 *ZSCAN* 命令的第一个参数总是一个数据库键。
- 而 *SCAN* 命令则不需要在第一个参数提供任何数据库键 —— 因为它迭代的是当前数据库中的所有数据库键。

SCAN 命令的基本用法

SCAN 命令是一个基于游标的迭代器（cursor based iterator）： *SCAN* 命令每次被调用之后， 都会向用户返回一个新的游标， 用户在下次迭代时需要使用这个新游标作为 *SCAN* 命令的游标参数， 以此来延续之前的迭代过程。

当 *SCAN* 命令的游标参数被设置为 `0` 时， 服务器将开始一次新的迭代， 而当服务器向用户返回值为 `0` 的游标时， 表示迭代已结束。

以下是一个 *SCAN* 命令的迭代过程示例：

```
redis 127.0.0.1:6379> scan 0
1) "17"
2)  1) "key:12"
    2) "key:8"
    3) "key:4"
    4) "key:14"
    5) "key:16"
    6) "key:17"
    7) "key:15"
    8) "key:10"
    9) "key:3"
    10) "key:7"
    11) "key:1"

redis 127.0.0.1:6379> scan 17
1) "0"
2) 1) "key:5"
   2) "key:18"
   3) "key:0"
   4) "key:2"
   5) "key:19"
   6) "key:13"
   7) "key:6"
   8) "key:9"
   9) "key:11"
```

在上面这个例子中，第一次迭代使用 `0` 作为游标，表示开始一次新的迭代。

第二次迭代使用的是第一次迭代时返回的游标，也即是命令回复第一个元素的值 —— `17` 。

从上面的示例可以看到，*SCAN* 命令的回复是一个包含两个元素的数组，第一个数组元素是用于进行下一次迭代的新游标，而第二个数组元素则是一个数组，这个数组中包含了所有被迭代的元素。

在第二次调用 *SCAN* 命令时，命令返回了游标 `0` ，这表示迭代已经结束，整个数据集（collection）已经被完整遍历过了。

以 `0` 作为游标开始一次新的迭代，一直调用 *SCAN* 命令，直到命令返回游标 `0` ，我们称这个过程为一次完整遍历（full iteration）。

SCAN 命令的保证（guarantees）

SCAN 命令，以及其他增量式迭代命令，在进行完整遍历的情况下可以为用户带来以下保证：从完整遍历开始直到完整遍历结束期间，一直存在于数据集内的所有元素都会被完整遍历返回；这意味着，如果有一个元素，它从遍历开始直到遍历结束期间都存在于被遍历的数据集当中，那么 *SCAN* 命令总会在某次迭代中将这个元素返回给用户。

然而因为增量式命令仅仅使用游标来记录迭代状态， 所以这些命令带有以下缺点：

- 同一个元素可能会被返回多次。 处理重复元素的工作交由应用程序负责， 比如说， 可以考虑将迭代返回的元素仅仅用于可以安全地重复执行多次的操作上。
- 如果一个元素是在迭代过程中被添加到数据集的， 又或者是在迭代过程中从数据集中被删除的， 那么这个元素可能会被返回， 也可能不会， 这是未定义的（undefined）。

SCAN 命令每次执行返回的元素数量

增量式迭代命令并不保证每次执行都返回某个给定数量的元素。

增量式命令甚至可能会返回零个元素， 但只要命令返回的游标不是 `0` ， 应用程序就不应该将迭代视作结束。

不过命令返回的元素数量总是符合一定规则的， 在实际中：

- 对于一个大数据集来说， 增量式迭代命令每次最多可能会返回数十个元素；
- 而对于一个足够小的数据集来说， 如果这个数据集的底层表示为编码数据结构（encoded data structure，适用于是小集合键、小哈希键和小有序集合键）， 那么增量迭代命令将在一次调用中返回数据集中的所有元素。

最后， 用户可以通过增量式迭代命令提供的 `COUNT` 选项来指定每次迭代返回元素的最大值。

COUNT 选项

虽然增量式迭代命令不保证每次迭代所返回的元素数量， 但我们可以使用 `COUNT` 选项， 对命令的行为进行一定程度上的调整。

基本上， `COUNT` 选项的作用就是让用户告知迭代命令， 在每次迭代中应该从数据集里返回多少元素。

虽然 `COUNT` 选项只是对增量式迭代命令的一种提示（hint）， 但是在大多数情况下， 这种提示都是有效的。

- `COUNT` 参数的默认值为 `10` 。
- 在迭代一个足够大的、由哈希表实现的数据库、集合键、哈希键或者有序集合键时， 如果用户没有使用 `MATCH` 选项， 那么命令返回的元素数量通常和 `COUNT` 选项指定的一样， 或者比 `COUNT` 选项指定的数量稍多一些。
- 在迭代一个编码为整数集合（intset，一个只由整数值构成的小集合）、 或者编码为压缩列表（ziplist，由不同值构成的一个小哈希或者一个小有序集合）时， 增量式迭代命令通常会无视 `COUNT` 选项指定的值， 在第一次迭代就将数据集包含的所有元素都返回给用户。

Note

并非每次迭代都要使用相同的 `COUNT` 值。

用户可以在每次迭代中按自己的需要随意改变 `COUNT` 值， 只要记得将上次迭代返回的游标用到下次迭代里面就可以了。

MATCH 选项

和 *KEYS* 命令一样， 增量式迭代命令也可以通过提供一个 glob 风格的模式参数， 让命令只返回和给定模式相匹配的元素， 这一点可以通过在执行增量式迭代命令时， 通过给定 `MATCH <pattern>` 参数来实现。

以下是一个使用 `MATCH` 选项进行迭代的示例：

```
redis 127.0.0.1:6379> sadd myset 1 2 3 foo foobar feelsgood
(integer) 6

redis 127.0.0.1:6379> sscan myset 0 match f*
1) "0"
2) 1) "foo"
   2) "feelsgood"
   3) "foobar"
```

需要注意的是， 对元素的模式匹配工作是在命令从数据集中取出元素之后， 向客户端返回元素之前的这段时间内进行的， 所以如果被迭代的数据集中只有少量元素和模式相匹配， 那么迭代命令或许会在多次执行中都不返回任何元素。

以下是这种情况的一个例子：

```
redis 127.0.0.1:6379> scan 0 MATCH *11*
1) "288"
2) 1) "key:911"

redis 127.0.0.1:6379> scan 288 MATCH *11*
1) "224"
2) (empty list or set)

redis 127.0.0.1:6379> scan 224 MATCH *11*
1) "80"
2) (empty list or set)

redis 127.0.0.1:6379> scan 80 MATCH *11*
1) "176"
2) (empty list or set)

redis 127.0.0.1:6379> scan 176 MATCH *11* COUNT 1000
1) "0"
2)  1) "key:611"
    2) "key:711"
    3) "key:118"
    4) "key:117"
    5) "key:311"
    6) "key:112"
    7) "key:111"
    8) "key:110"
    9) "key:113"
   10) "key:211"
   11) "key:411"
   12) "key:115"
   13) "key:116"
   14) "key:114"
   15) "key:119"
   16) "key:811"
   17) "key:511"
   18) "key:11"
```

如你所见， 以上的大部分迭代都不返回任何元素。

在最后一次迭代， 我们通过将 `COUNT` 选项的参数设置为 `1000` ， 强制命令为本次迭代扫描更多元素， 从而使得命令返回的元素也变多了。

并发执行多个迭代

在同一时间， 可以有任意多个客户端对同一数据集进行迭代， 客户端每次执行迭代都需要传入一个游标， 并在迭代执行之后获得一个新的游标， 而这个游标就包含了迭代的所有状态， 因此， 服务器无须为迭代记录任何状态。

中途停止迭代

因为迭代的所有状态都保存在游标里面， 而服务器无须为迭代保存任何状态， 所以客户端可以在中途停止一个迭代， 而无须对服务器进行任何通知。

即使有任意数量的迭代在中途停止， 也不会产生任何问题。

使用错误的游标进行增量式迭代

使用间断的（broken）、负数、超出范围或者其他非正常的游标来执行增量式迭代并不会造成服务器崩溃， 但可能会让命令产生未定义的行为。

未定义行为指的是， 增量式命令对返回值所做的保证可能会不再为真。

只有两种游标是合法的：

1. 在开始一个新的迭代时， 游标必须为 `0` 。
2. 增量式迭代命令在执行之后返回的， 用于延续（continue）迭代过程的游标。

迭代终结的保证

增量式迭代命令所使用的算法只保证在数据集的大小有界（bounded）的情况下， 迭代才会停止， 换句话说， 如果被迭代数据集的大小不断地增长的话， 增量式迭代命令可能永远也无法完成一次完整迭代。

从直觉上可以看出， 当一个数据集不断地变大时， 想要访问这个数据集中的所有元素就需要做越来越多的工作， 能否结束一个迭代取决于用户执行迭代的速度是否比数据集增长的速度更快。

可用版本：

> >= 2.8.0

时间复杂度：

> 增量式迭代命令每次执行的复杂度为 O(1)， 对数据集进行一次完整迭代的复杂度为 O(N)， 其中 N 为数据集中的元素数量。

返回值：

> *SCAN* 命令、 *SSCAN* 命令、 *HSCAN* 命令和 *ZSCAN* 命令都返回一个包含两个元素的 multi-bulk 回复： 回复的第一个元素是字符串表示的无符号 64 位整数（游标）， 回复的第二个元素是另一个 multi-bulk 回复， 这个 multi-bulk 回复包含了本次被迭代的元素。 > > *SCAN* 命令返回的每个元素都是一个数据库键。 > > *SSCAN* 命令返回的每个元素都是一个集合成员。 > > *HSCAN* 命令返回的每个元素都是一个键值对，一个键值对由一个键和一个值组成。 > > *ZSCAN* 命令返回的每个元素都是一个有序集合元素，一个有序集合元素由一个成员（member）和一个分值（score）组成。

String（字符串）

APPEND

APPEND key value

如果 `key` 已经存在并且是一个字符串， APPEND 命令将 `value` 追加到 `key` 原来的值的末尾。

如果 `key` 不存在， APPEND 就简单地将给定 `key` 设为 `value` ，就像执行 `SET key value` 一样。

可用版本：

>= 2.0.0

时间复杂度：

平摊O(1)

返回值：

追加 `value` 之后， `key` 中字符串的长度。

```
# 对不存在的 key 执行 APPEND

redis> EXISTS myphone               # 确保 myphone 不存在
(integer) 0

redis> APPEND myphone "nokia"       # 对不存在的 key 进行 APPEND ，等
(integer) 5                         # 字符长度

# 对已存在的字符串进行 APPEND

redis> APPEND myphone " - 1110"     # 长度从 5 个字符增加到 12 个字符
(integer) 12

redis> GET myphone
"nokia - 1110"
```

模式：时间序列(Time series)

APPEND 可以为一系列定长(fixed-size)数据(sample)提供一种紧凑的表示方式，通常称之为时间序列。

每当一个新数据到达的时候，执行以下命令：

```
APPEND timeseries "fixed-size sample"
```

然后可以通过以下的方式访问时间序列的各项属性：

- *STRLEN* 给出时间序列中数据的数量
- *GETRANGE* 可以用于随机访问。只要有相关的时间信息的话，我们就可以在 Redis 2.6 中使用 Lua 脚本和 *GETRANGE* 命令实现二分查找。
- *SETRANGE* 可以用于覆盖或修改已存在的的时间序列。

这个模式的唯一缺陷是我们只能增长时间序列，而不能对时间序列进行缩短，因为 Redis 目前还没有对字符串进行修剪(tirm)的命令，但是，不管怎么说，这个模式的储存方式还是可以节省下大量的空间。

Note

可以考虑使用 UNIX 时间戳作为时间序列的键名，这样一来，可以避免单个 `key` 因为保存过大的时间序列而占用大量内存，另一方面，也可以节省下大量命名空间。

下面是一个时间序列的例子：

```
redis> APPEND ts "0043"
(integer) 4

redis> APPEND ts "0035"
(integer) 8

redis> GETRANGE ts 0 3
"0043"

redis> GETRANGE ts 4 7
"0035"
```

BITCOUNT

BITCOUNT key [start] [end]

计算给定字符串中，被设置为 `1` 的比特位的数量。

一般情况下，给定的整个字符串都会被进行计数，通过指定额外的 `start` 或 `end` 参数，可以让计数只在特定的位上进行。

`start` 和 `end` 参数的设置和 *GETRANGE* 命令类似，都可以使用负数值：比如 `-1` 表示最后一个字节， `-2` 表示倒数第二个字节，以此类推。

不存在的 `key` 被当成是空字符串来处理，因此对一个不存在的 `key` 进行 `BITCOUNT` 操作，结果为 `0` 。

可用版本：

>= 2.6.0

时间复杂度：

O(N)

返回值：

被设置为 `1` 的位的数量。

```
redis> BITCOUNT bits
(integer) 0

redis> SETBIT bits 0 1          # 0001
(integer) 0

redis> BITCOUNT bits
(integer) 1

redis> SETBIT bits 3 1          # 1001
(integer) 0

redis> BITCOUNT bits
(integer) 2
```

模式：使用 bitmap 实现用户上线次数统计

Bitmap 对于一些特定类型的计算非常有效。

假设现在我们希望记录自己网站上的用户的上线频率，比如说，计算用户 A 上线了多少天，用户 B 上线了多少天，诸如此类，以此作为数据，从而决定让哪些用户参加 beta 测试等活动 —— 这个模式可以使用 *SETBIT* 和 *BITCOUNT* 来实现。

比如说，每当用户在某一天上线的时候，我们就使用 *SETBIT*，以用户名作为 `key`，将那天所代表的网站的上线日作为 `offset` 参数，并将这个 `offset` 上的为设置为 `1`。

举个例子，如果今天是网站上线的第 100 天，而用户 peter 在今天阅览过网站，那么执行命令 `SETBIT peter 100 1`；如果明天 peter 也继续阅览网站，那么执行命令 `SETBIT peter 101 1`，以此类推。

当要计算 peter 总共以来的上线次数时，就使用 *BITCOUNT* 命令：执行 `BITCOUNT peter`，得出的结果就是 peter 上线的总天数。

更详细的实现可以参考博文(墙外) Fast, easy, realtime metrics using Redis bitmaps 。

性能

前面的上线次数统计例子，即使运行 10 年，占用的空间也只是每个用户 10*365 比特位(bit)，也即是每个用户 456 字节。对于这种大小的数据来说， *BITCOUNT* 的处理速度就像 *GET* 和 *INCR* 这种 O(1) 复杂度的操作一样快。

如果你的 bitmap 数据非常大，那么可以考虑使用以下两种方法：

1. 将一个大的 bitmap 分散到不同的 key 中，作为小的 bitmap 来处理。使用 Lua 脚本可以很方便地完成这一工作。
2. 使用 *BITCOUNT* 的 `start` 和 `end` 参数，每次只对所需的部分位进行计算，将位的累积工作(accumulating)放到客户端进行，并且对结果进行缓存 (caching)。

BITOP

BITOP operation destkey key [key ...]

对一个或多个保存二进制位的字符串 `key` 进行位元操作，并将结果保存到 `destkey` 上。

`operation` 可以是 `AND` 、 `OR` 、 `NOT` 、 `XOR` 这四种操作中的任意一种：

- `BITOP AND destkey key [key ...]` ，对一个或多个 `key` 求逻辑并，并将结果保存到 `destkey` 。
- `BITOP OR destkey key [key ...]` ，对一个或多个 `key` 求逻辑或，并将结果保存到 `destkey` 。
- `BITOP XOR destkey key [key ...]` ，对一个或多个 `key` 求逻辑异或，并将结果保存到 `destkey` 。
- `BITOP NOT destkey key` ，对给定 `key` 求逻辑非，并将结果保存到 `destkey` 。

除了 `NOT` 操作之外，其他操作都可以接受一个或多个 `key` 作为输入。

处理不同长度的字符串

当 BITOP 处理不同长度的字符串时，较短的那个字符串所缺少的部分会被看作 `0` 。

空的 `key` 也被看作是包含 `0` 的字符串序列。

可用版本：

>= 2.6.0

时间复杂度：

O(N)

返回值：

保存到 `destkey` 的字符串的长度，和输入 `key` 中最长的字符串长度相等。

Note

BITOP 的复杂度为 O(N) ，当处理大型矩阵(matrix)或者进行大数据量的统计时，最好将任务指派到附属节点(slave)进行，避免阻塞主节点。

```
redis> SETBIT bits-1 0 1        # bits-1 = 1001
(integer) 0

redis> SETBIT bits-1 3 1
(integer) 0

redis> SETBIT bits-2 0 1        # bits-2 = 1011
(integer) 0

redis> SETBIT bits-2 1 1
(integer) 0

redis> SETBIT bits-2 3 1
(integer) 0

redis> BITOP AND and-result bits-1 bits-2
(integer) 1

redis> GETBIT and-result 0      # and-result = 1001
(integer) 1

redis> GETBIT and-result 1
(integer) 0

redis> GETBIT and-result 2
(integer) 0

redis> GETBIT and-result 3
(integer) 1
```

DECR

DECR key

将 `key` 中储存的数字值减一。

如果 `key` 不存在，那么 `key` 的值会先被初始化为 `0` ，然后再执行 DECR 操作。

如果值包含错误的类型，或字符串类型的值不能表示为数字，那么返回一个错误。

本操作的值限制在 64 位(bit)有符号数字表示之内。

关于递增(increment) / 递减(decrement)操作的更多信息，请参见 *INCR* 命令。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

执行 DECR 命令之后 `key` 的值。

```
# 对存在的数字值 key 进行 DECR

redis> SET failure_times 10
OK

redis> DECR failure_times
(integer) 9

# 对不存在的 key 值进行 DECR

redis> EXISTS count
(integer) 0

redis> DECR count
(integer) -1

# 对存在但不是数值的 key 进行 DECR

redis> SET company YOUR_CODE_SUCKS.LLC
OK

redis> DECR company
(error) ERR value is not an integer or out of range
```

DECRBY

DECRBY key decrement

将 `key` 所储存的值减去减量 `decrement` 。

如果 `key` 不存在，那么 `key` 的值会先被初始化为 `0` ，然后再执行 DECRBY 操作。

如果值包含错误的类型，或字符串类型的值不能表示为数字，那么返回一个错误。

本操作的值限制在 64 位(bit)有符号数字表示之内。

关于更多递增(increment) / 递减(decrement)操作的更多信息，请参见 *INCR* 命令。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

减去 `decrement` 之后， `key` 的值。

```
# 对已存在的 key 进行 DECRBY

redis> SET count 100
OK

redis> DECRBY count 20
(integer) 80

# 对不存在的 key 进行DECRBY

redis> EXISTS pages
(integer) 0

redis> DECRBY pages 10
(integer) -10
```

GET

GET key

返回 `key` 所关联的字符串值。

如果 `key` 不存在那么返回特殊值 `nil` 。

假如 `key` 储存的值不是字符串类型，返回一个错误，因为 GET 只能用于处理字符串值。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

当 `key` 不存在时，返回 `nil` ，否则，返回 `key` 的值。如果 `key` 不是字符串类型，那么返回一个错误。

```
# 对不存在的 key 或字符串类型 key 进行 GET

redis> GET db
(nil)

redis> SET db redis
OK

redis> GET db
"redis"

# 对不是字符串类型的 key 进行 GET

redis> DEL db
(integer) 1

redis> LPUSH db redis mongodb mysql
(integer) 3

redis> GET db
(error) ERR Operation against a key holding the wrong kind of value
```

GETBIT

GETBIT key offset

对 `key` 所储存的字符串值，获取指定偏移量上的位(bit)。

当 `offset` 比字符串值的长度大，或者 `key` 不存在时，返回 `0` 。

可用版本：

>= 2.2.0

时间复杂度：

O(1)

返回值：

字符串值指定偏移量上的位(bit)。

```
# 对不存在的 key 或者不存在的 offset 进行 GETBIT， 返回 0

redis> EXISTS bit
(integer) 0

redis> GETBIT bit 10086
(integer) 0

# 对已存在的 offset 进行 GETBIT

redis> SETBIT bit 10086 1
(integer) 0

redis> GETBIT bit 10086
(integer) 1
```

GETRANGE

GETRANGE key start end

返回 `key` 中字符串值的子字符串，字符串的截取范围由 `start` 和 `end` 两个偏移量决定(包括 `start` 和 `end` 在内)。

负数偏移量表示从字符串最后开始计数， `-1` 表示最后一个字符， `-2` 表示倒数第二个，以此类推。

GETRANGE 通过保证子字符串的值域(range)不超过实际字符串的值域来处理超出范围的值域请求。

Note

在 <= 2.0 的版本里，GETRANGE 被叫作 SUBSTR。

可用版本：

>= 2.4.0

时间复杂度：

O(N)， `N` 为要返回的字符串的长度。复杂度最终由字符串的返回值长度决定，但因为从已有字符串中取出子字符串的操作非常廉价(cheap)，所以对于长度不大的字符串，该操作的复杂度也可看作O(1)。

返回值：

截取得出的子字符串。

```
redis> SET greeting "hello, my friend"
OK

redis> GETRANGE greeting 0 4           # 返回索引0-4的字符，包括4。
"hello"

redis> GETRANGE greeting -1 -5         # 不支持回绕操作
""

redis> GETRANGE greeting -3 -1         # 负数索引
"end"

redis> GETRANGE greeting 0 -1          # 从第一个到最后一个
"hello, my friend"

redis> GETRANGE greeting 0 1008611     # 值域范围不超过实际字符串，超过部
"hello, my friend"
```

GETSET

GETSET key value

将给定 `key` 的值设为 `value` ，并返回 `key` 的旧值(old value)。

当 `key` 存在但不是字符串类型时，返回一个错误。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

返回给定 `key` 的旧值。当 `key` 没有旧值时，也即是， `key` 不存在时，返回 `nil` 。

```
redis> GETSET db mongodb    # 没有旧值，返回 nil
(nil)

redis> GET db
"mongodb"

redis> GETSET db redis      # 返回旧值 mongodb
"mongodb"

redis> GET db
"redis"
```

模式

GETSET 可以和 *INCR* 组合使用，实现一个有原子性(atomic)复位操作的计数器(counter)。

举例来说，每次当某个事件发生时，进程可能对一个名为 `mycount` 的 `key` 调用 *INCR* 操作，通常我们还要在一个原子时间内同时完成获得计数器的值和将计数器值复位为 `0` 两个操作。

可以用命令 `GETSET mycounter 0` 来实现这一目标。

```
redis> INCR mycount
(integer) 11

redis> GETSET mycount 0  # 一个原子内完成 GET mycount 和 SET mycount
"11"

redis> GET mycount       # 计数器被重置
"0"
```

INCR

INCR key

将 `key` 中储存的数字值增一。

如果 `key` 不存在，那么 `key` 的值会先被初始化为 `0` ，然后再执行 INCR 操作。

如果值包含错误的类型，或字符串类型的值不能表示为数字，那么返回一个错误。

本操作的值限制在 64 位(bit)有符号数字表示之内。

Note

这是一个针对字符串的操作，因为 Redis 没有专用的整数类型，所以 key 内储存的字符串被解释为十进制 64 位有符号整数来执行 INCR 操作。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

执行 INCR 命令之后 `key` 的值。

```
redis> SET page_view 20
OK

redis> INCR page_view
(integer) 21

redis> GET page_view    # 数字值在 Redis 中以字符串的形式保存
"21"
```

模式：计数器

计数器是 Redis 的原子性自增操作可实现的最直观的模式了，它的想法相当简单：每当某个操作发生时，向 Redis 发送一个 INCR 命令。

比如在一个 web 应用程序中，如果想知道用户在一年中每天的点击量，那么只要将用户 ID 以及相关的日期信息作为键，并在每次用户点击页面时，执行一次自增操作即可。

比如用户名是 `peter` ，点击时间是 2012 年 3 月 22 日，那么执行命令 `INCR peter::2012.3.22` 。

可以用以下几种方式扩展这个简单的模式：

- 可以通过组合使用 INCR 和 *EXPIRE* ，来达到只在规定的生存时间内进行计数(counting)的目的。
- 客户端可以通过使用 *GETSET* 命令原子性地获取计数器的当前值并将计数器清零，更多信息请参考 *GETSET* 命令。
- 使用其他自增/自减操作，比如 *DECR* 和 *INCRBY* ，用户可以通过执行不同的操作增加或减少计数器的值，比如在游戏中的记分器就可能用到这些命令。

模式：限速器

限速器是特殊化的计算器，它用于限制一个操作可以被执行的速率(rate)。

限速器的典型用法是限制公开 API 的请求次数，以下是一个限速器实现示例，它将 API 的最大请求数限制在每个 IP 地址每秒钟十个之内：

```
FUNCTION LIMIT_API_CALL(ip)
ts = CURRENT_UNIX_TIME()
keyname = ip+":"+ts
current = GET(keyname)

IF current != NULL AND current > 10 THEN
    ERROR "too many requests per second"
END

IF current == NULL THEN
    MULTI
        INCR(keyname, 1)
        EXPIRE(keyname, 1)
    EXEC
ELSE
    INCR(keyname, 1)
END

PERFORM_API_CALL()
```

这个实现每秒钟为每个 IP 地址使用一个不同的计数器，并用 *EXPIRE* 命令设置生存时间(这样 Redis 就会负责自动删除过期的计数器)。

注意，我们使用事务打包执行 *INCR* 命令和 *EXPIRE* 命令，避免引入竞争条件，保证每次调用 API 时都可以正确地对计数器进行自增操作并设置生存时间。

以下是另一个限速器实现：

```
FUNCTION LIMIT_API_CALL(ip):
current = GET(ip)
IF current != NULL AND current > 10 THEN
    ERROR "too many requests per second"
ELSE
    value = INCR(ip)
    IF value == 1 THEN
        EXPIRE(ip,1)
    END
    PERFORM_API_CALL()
END
```

这个限速器只使用单个计数器，它的生存时间为一秒钟，如果在一秒钟内，这个计数器的值大于 `10` 的话，那么访问就会被禁止。

这个新的限速器在思路方面是没有问题的，但它在实现方面不够严谨，如果我们仔细观察一下的话，就会发现在 *INCR* 和 *EXPIRE* 之间存在着一个竞争条件，假如客户端在执行 *INCR* 之后，因为某些原因(比如客户端失败)而忘记设置 *EXPIRE* 的话，那么这个计数器就会一直存在下去，造成每个用户只能访问 `10` 次，噢，这简直是个灾难！

要消灭这个实现中的竞争条件，我们可以将它转化为一个 Lua 脚本，并放到 Redis 中运行(这个方法仅限于 Redis 2.6 及以上的版本)：

```
local current
current = redis.call("incr",KEYS[1])
if tonumber(current) == 1 then
    redis.call("expire",KEYS[1],1)
end
```

通过将计数器作为脚本放到 Redis 上运行，我们保证了 *INCR* 和 *EXPIRE* 两个操作的原子性，现在这个脚本实现不会引入竞争条件，它可以运作的很好。

关于在 Redis 中运行 Lua 脚本的更多信息，请参考 *EVAL* 命令。

还有另一种消灭竞争条件的方法，就是使用 Redis 的列表结构来代替 *INCR* 命令，这个方法无须脚本支持，因此它在 Redis 2.6 以下的版本也可以运行得很好：

```
FUNCTION LIMIT_API_CALL(ip)
current = LLEN(ip)
IF current > 10 THEN
    ERROR "too many requests per second"
ELSE
    IF EXISTS(ip) == FALSE
        MULTI
            RPUSH(ip,ip)
            EXPIRE(ip,1)
        EXEC
    ELSE
        RPUSHX(ip,ip)
    END
    PERFORM_API_CALL()
END
```

新的限速器使用了列表结构作为容器， *LLEN* 用于对访问次数进行检查，一个事务包裹着 *RPUSH* 和 *EXPIRE* 两个命令，用于在第一次执行计数时创建列表，并正确设置地设置过期时间，最后， *RPUSHX* 在后续的计数操作中进行增加操作。

INCRBY

INCRBY key increment

将 `key` 所储存的值加上增量 `increment` 。

如果 `key` 不存在，那么 `key` 的值会先被初始化为 `0` ，然后再执行 INCRBY 命令。

如果值包含错误的类型，或字符串类型的值不能表示为数字，那么返回一个错误。

本操作的值限制在 64 位(bit)有符号数字表示之内。

关于递增(increment) / 递减(decrement)操作的更多信息，参见 *INCR* 命令。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

加上 `increment` 之后， `key` 的值。

```
# key 存在且是数字值

redis> SET rank 50
OK

redis> INCRBY rank 20
(integer) 70

redis> GET rank
"70"

# key 不存在时

redis> EXISTS counter
(integer) 0

redis> INCRBY counter 30
(integer) 30

redis> GET counter
"30"

# key 不是数字值时

redis> SET book "long long ago..."
OK

redis> INCRBY book 200
(error) ERR value is not an integer or out of range
```

INCRBYFLOAT

INCRBYFLOAT key increment

为 `key` 中所储存的值加上浮点数增量 `increment` 。

如果 `key` 不存在，那么 INCRBYFLOAT 会先将 `key` 的值设为 `0` ，再执行加法操作。

如果命令执行成功，那么 `key` 的值会被更新为（执行加法之后的）新值，并且新值会以字符串的形式返回给调用者。

无论是 `key` 的值，还是增量 `increment` ，都可以使用像 `2.0e7` 、 `3e5` 、 `90e-2` 那样的指数符号(exponential notation)来表示，但是，执行 **INCRBYFLOAT** 命令之后的值总是以同样的形式储存，也即是，它们总是由一个数字，一个（可选的）小数点和一个任意位的小数部分组成（比如 `3.14` 、 `69.768` ，诸如此类)，小数部分尾随的 `0` 会被移除，如果有需要的话，还会将浮点数改为整数（比如 `3.0` 会被保存成 `3` ）。

除此之外，无论加法计算所得的浮点数的实际精度有多长， INCRBYFLOAT 的计算结果也最多只能表示小数点的后十七位。

当以下任意一个条件发生时，返回一个错误：

- `key` 的值不是字符串类型(因为 Redis 中的数字和浮点数都以字符串的形式保存，所以它们都属于字符串类型）
- `key` 当前的值或者给定的增量 `increment` 不能解释(parse)为双精度浮点数(double precision floating point number）

可用版本：

>= 2.6.0

时间复杂度：

O(1)

返回值：

执行命令之后 `key` 的值。

```
# 值和增量都不是指数符号

redis> SET mykey 10.50
OK

redis> INCRBYFLOAT mykey 0.1
"10.6"

# 值和增量都是指数符号

redis> SET mykey 314e-2
OK

redis> GET mykey                # 用 SET 设置的值可以是指数符号
"314e-2"

redis> INCRBYFLOAT mykey 0      # 但执行 INCRBYFLOAT 之后格式会被改成非
"3.14"

# 可以对整数类型执行

redis> SET mykey 3
OK

redis> INCRBYFLOAT mykey 1.1
"4.1"

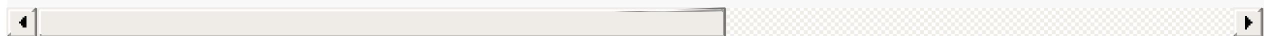
# 后跟的 0 会被移除

redis> SET mykey 3.0
OK

redis> GET mykey                                    # SET 设置的值小
"3.0"

redis> INCRBYFLOAT mykey 1.000000000000000000000    # 但 INCRBYFLOA
"4"

redis> GET mykey
"4"
```



MGET

MGET key [key ...]

返回所有(一个或多个)给定 `key` 的值。

如果给定的 `key` 里面，有某个 `key` 不存在，那么这个 `key` 返回特殊值 `nil` 。因此，该命令永不失败。

可用版本：

>= 1.0.0

时间复杂度:

O(N) , `N` 为给定 `key` 的数量。

返回值：

一个包含所有给定 `key` 的值的列表。

```
redis> SET redis redis.com
OK

redis> SET mongodb mongodb.org
OK

redis> MGET redis mongodb
1) "redis.com"
2) "mongodb.org"

redis> MGET redis mongodb mysql     # 不存在的 mysql 返回 nil
1) "redis.com"
2) "mongodb.org"
3) (nil)
```

MSET

MSET key value [key value ...]

同时设置一个或多个 `key-value` 对。

如果某个给定 `key` 已经存在，那么 MSET 会用新值覆盖原来的旧值，如果这不是你所希望的效果，请考虑使用 *MSETNX* 命令：它只会在所有给定 `key` 都不存在的情况下进行设置操作。

MSET 是一个原子性(atomic)操作，所有给定 `key` 都会在同一时间内被设置，某些给定 `key` 被更新而另一些给定 `key` 没有改变的情况，不可能发生。

可用版本：

>= 1.0.1

时间复杂度：

O(N)， `N` 为要设置的 `key` 数量。

返回值：

总是返回 `OK` (因为 `MSET` 不可能失败)

```
redis> MSET date "2012.3.30" time "11:00 a.m." weather "sunny"
OK

redis> MGET date time weather
1) "2012.3.30"
2) "11:00 a.m."
3) "sunny"

# MSET 覆盖旧值例子

redis> SET google "google.hk"
OK

redis> MSET google "google.com"
OK

redis> GET google
"google.com"
```

MSETNX

MSETNX key value [key value ...]

同时设置一个或多个 `key-value` 对，当且仅当所有给定 `key` 都不存在。

即使只有一个给定 `key` 已存在， MSETNX 也会拒绝执行所有给定 `key` 的设置操作。

MSETNX 是原子性的，因此它可以用作设置多个不同 `key` 表示不同字段(field)的唯一性逻辑对象(unique logic object)，所有字段要么全被设置，要么全不被设置。

可用版本：

>= 1.0.1

时间复杂度：

O(N)， `N` 为要设置的 `key` 的数量。

返回值：

当所有 `key` 都成功设置，返回 `1` 。如果所有给定 `key` 都设置失败(至少有一个 `key` 已经存在)，那么返回 `0` 。

```
# 对不存在的 key 进行 MSETNX

redis> MSETNX rmdbs "MySQL" nosql "MongoDB" key-value-store "redis'
(integer) 1

redis> MGET rmdbs nosql key-value-store
1) "MySQL"
2) "MongoDB"
3) "redis"

# MSET 的给定 key 当中有已存在的 key

redis> MSETNX rmdbs "Sqlite" language "python"  # rmdbs 键已经存在，扌
(integer) 0

redis> EXISTS language                          # 因为 MSET 是原子性扌
(integer) 0

redis> GET rmdbs                                # rmdbs 也没有被修改
"MySQL"
```

PSETEX

PSETEX key milliseconds value

这个命令和 *SETEX* 命令相似，但它以毫秒为单位设置 `key` 的生存时间，而不是像 *SETEX* 命令那样，以秒为单位。

可用版本：

>= 2.6.0

时间复杂度：

O(1)

返回值：

设置成功时返回 `OK` 。

```
redis> PSETEX mykey 1000 "Hello"
OK

redis> PTTL mykey
(integer) 999

redis> GET mykey
"Hello"
```

SET

SET key value [EX seconds] [PX milliseconds] [NX|XX]

将字符串值 `value` 关联到 `key` 。

如果 `key` 已经持有其他值， SET 就覆写旧值，无视类型。

对于某个原本带有生存时间（TTL）的键来说， 当 *SET* 命令成功在这个键上执行时， 这个键原有的 TTL 将被清除。

可选参数

从 Redis 2.6.12 版本开始， *SET* 命令的行为可以通过一系列参数来修改：

- `EX second` ：设置键的过期时间为 `second` 秒。 `SET key value EX second` 效果等同于 `SETEX key second value` 。
- `PX millisecond` ：设置键的过期时间为 `millisecond` 毫秒。 `SET key value PX millisecond` 效果等同于 `PSETEX key millisecond value` 。
- `NX` ：只在键不存在时，才对键进行设置操作。 `SET key value NX` 效果等同于 `SETNX key value` 。
- `XX` ：只在键已经存在时，才对键进行设置操作。

Note

因为 *SET* 命令可以通过参数来实现和 *SETNX* 、 *SETEX* 和 *PSETEX* 三个命令的效果，所以将来的 Redis 版本可能会废弃并最终移除 *SETNX* 、 *SETEX* 和 *PSETEX* 这三个命令。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

在 Redis 2.6.12 版本以前， *SET* 命令总是返回 `OK` 。

从 Redis 2.6.12 版本开始， *SET* 在设置操作成功完成时，才返回 `OK` 。如果设置了 `NX` 或者 `XX` ，但因为条件没达到而造成设置操作未执行，那么命令返回空批量回复（NULL Bulk Reply）。

```
# 对不存在的键进行设置

redis 127.0.0.1:6379> SET key "value"
```

```
OK

redis 127.0.0.1:6379> GET key
"value"

# 对已存在的键进行设置

redis 127.0.0.1:6379> SET key "new-value"
OK

redis 127.0.0.1:6379> GET key
"new-value"

# 使用 EX 选项

redis 127.0.0.1:6379> SET key-with-expire-time "hello" EX 10086
OK

redis 127.0.0.1:6379> GET key-with-expire-time
"hello"

redis 127.0.0.1:6379> TTL key-with-expire-time
(integer) 10069

# 使用 PX 选项

redis 127.0.0.1:6379> SET key-with-pexpire-time "moto" PX 123321
OK

redis 127.0.0.1:6379> GET key-with-pexpire-time
"moto"

redis 127.0.0.1:6379> PTTL key-with-pexpire-time
(integer) 111939

# 使用 NX 选项

redis 127.0.0.1:6379> SET not-exists-key "value" NX
OK       # 键不存在，设置成功

redis 127.0.0.1:6379> GET not-exists-key
"value"

redis 127.0.0.1:6379> SET not-exists-key "new-value" NX
(nil)   # 键已经存在，设置失败

redis 127.0.0.1:6379> GEt not-exists-key
"value" # 维持原值不变

# 使用 XX 选项

redis 127.0.0.1:6379> EXISTS exists-key
(integer) 0
```

```
redis 127.0.0.1:6379> SET exists-key "value" XX
(nil)   # 因为键不存在，设置失败

redis 127.0.0.1:6379> SET exists-key "value"
OK      # 先给键设置一个值

redis 127.0.0.1:6379> SET exists-key "new-value" XX
OK      # 设置新值成功

redis 127.0.0.1:6379> GET exists-key
"new-value"

# NX 或 XX 可以和 EX 或者 PX 组合使用

redis 127.0.0.1:6379> SET key-with-expire-and-NX "hello" EX 10086 N
OK

redis 127.0.0.1:6379> GET key-with-expire-and-NX
"hello"

redis 127.0.0.1:6379> TTL key-with-expire-and-NX
(integer) 10063

redis 127.0.0.1:6379> SET key-with-pexpire-and-XX "old value"
OK

redis 127.0.0.1:6379> SET key-with-pexpire-and-XX "new value" PX 12
OK

redis 127.0.0.1:6379> GET key-with-pexpire-and-XX
"new value"

redis 127.0.0.1:6379> PTTL key-with-pexpire-and-XX
(integer) 112999

# EX 和 PX 可以同时出现，但后面给出的选项会覆盖前面给出的选项

redis 127.0.0.1:6379> SET key "value" EX 1000 PX 5000000
OK

redis 127.0.0.1:6379> TTL key
(integer) 4993  # 这是 PX 参数设置的值

redis 127.0.0.1:6379> SET another-key "value" PX 5000000 EX 1000
OK

redis 127.0.0.1:6379> TTL another-key
(integer) 997   # 这是 EX 参数设置的值
```

使用模式

命令 `SET resource-name anystring NX EX max-lock-time` 是一种在 Redis 中实现锁的简单方法。

客户端执行以上的命令：

- 如果服务器返回 `OK` ，那么这个客户端获得锁。
- 如果服务器返回 `NIL` ，那么客户端获取锁失败，可以在稍后再重试。

设置的过期时间到达之后，锁将自动释放。

可以通过以下修改，让这个锁实现更健壮：

- 不使用固定的字符串作为键的值，而是设置一个不可猜测（non-guessable）的长随机字符串，作为口令串（token）。
- 不使用 *DEL* 命令来释放锁，而是发送一个 Lua 脚本，这个脚本只在客户端传入的值和键的口令串相匹配时，才对键进行删除。

这两个改动可以防止持有过期锁的客户端误删现有锁的情况出现。

以下是一个简单的解锁脚本示例：

```
if redis.call("get",KEYS[1]) == ARGV[1]
then
    return redis.call("del",KEYS[1])
else
    return 0
end
```

这个脚本可以通过 `EVAL ...script... 1 resource-name token-value` 命令来调用。

SETBIT

SETBIT key offset value

对 `key` 所储存的字符串值，设置或清除指定偏移量上的位(bit)。

位的设置或清除取决于 `value` 参数，可以是 `0` 也可以是 `1` 。

当 `key` 不存在时，自动生成一个新的字符串值。

字符串会进行伸展(grown)以确保它可以将 `value` 保存在指定的偏移量上。当字符串值进行伸展时，空白位置以 `0` 填充。

`offset` 参数必须大于或等于 `0` ，小于 2^32 (bit 映射被限制在 512 MB 之内)。

Warning

对使用大的 `offset` 的 SETBIT 操作来说，内存分配可能造成 Redis 服务器被阻塞。具体参考 *SETRANGE* 命令，warning(警告)部分。

可用版本：

>= 2.2.0

时间复杂度:

O(1)

返回值：

指定偏移量原来储存的位。

```
redis> SETBIT bit 10086 1
(integer) 0

redis> GETBIT bit 10086
(integer) 1

redis> GETBIT bit 100   # bit 默认被初始化为 0
(integer) 0
```

SETEX

SETEX key seconds value

将值 `value` 关联到 `key` ，并将 `key` 的生存时间设为 `seconds` (以秒为单位)。

如果 `key` 已经存在， SETEX 命令将覆写旧值。

这个命令类似于以下两个命令：

```
SET key value
EXPIRE key seconds  # 设置生存时间
```

不同之处是， SETEX 是一个原子性(atomic)操作，关联值和设置生存时间两个动作会在同一时间内完成，该命令在 Redis 用作缓存时，非常实用。

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

设置成功时返回 `OK` 。当 `seconds` 参数不合法时，返回一个错误。

```
# 在 key 不存在时进行 SETEX

redis> SETEX cache_user_id 60 10086
OK

redis> GET cache_user_id  # 值
"10086"

redis> TTL cache_user_id  # 剩余生存时间
(integer) 49

# key 已经存在时，SETEX 覆盖旧值

redis> SET cd "timeless"
OK

redis> SETEX cd 3000 "goodbye my love"
OK

redis> GET cd
"goodbye my love"

redis> TTL cd
(integer) 2997
```

SETNX

SETNX key value

将 `key` 的值设为 `value` ，当且仅当 `key` 不存在。

若给定的 `key` 已经存在，则 SETNX 不做任何动作。

SETNX 是『SET if Not eXists』(如果不存在，则 SET)的简写。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

设置成功，返回 `1` 。设置失败，返回 `0` 。

```
redis> EXISTS job                # job 不存在
(integer) 0

redis> SETNX job "programmer"    # job 设置成功
(integer) 1

redis> SETNX job "code-farmer"   # 尝试覆盖 job ，失败
(integer) 0

redis> GET job                   # 没有被覆盖
"programmer"
```

SETRANGE

SETRANGE key offset value

用 `value` 参数覆写(overwrite)给定 `key` 所储存的字符串值，从偏移量 `offset` 开始。

不存在的 `key` 当作空白字符串处理。

SETRANGE 命令会确保字符串足够长以便将 `value` 设置在指定的偏移量上，如果给定 `key` 原来储存的字符串长度比偏移量小(比如字符串只有 `5` 个字符长，但你设置的 `offset` 是 `10`)，那么原字符和偏移量之间的空白将用零字节(zerobytes, `"\x00"`)来填充。

注意你能使用的最大偏移量是 2^29-1(536870911) ，因为 Redis 字符串的大小被限制在 512 兆(megabytes)以内。如果你需要使用比这更大的空间，你可以使用多个 `key` 。

Warning

当生成一个很长的字符串时，Redis 需要分配内存空间，该操作有时候可能会造成服务器阻塞(block)。在2010年的Macbook Pro上，设置偏移量为 536870911(512MB 内存分配)，耗费约 300 毫秒， 设置偏移量为 134217728(128MB 内存分配)，耗费约 80 毫秒，设置偏移量 33554432(32MB 内存分配)，耗费约 30 毫秒，设置偏移量为 8388608(8MB 内存分配)，耗费约 8 毫秒。 注意若首次内存分配成功之后，再对同一个 `key` 调用 SETRANGE 操作，无须再重新内存。

可用版本：

>= 2.2.0

时间复杂度：

对小(small)的字符串，平摊复杂度O(1)。(关于什么字符串是"小"的，请参考 *APPEND* 命令)否则为O(M)， `M` 为 `value` 参数的长度。

返回值：

被 SETRANGE 修改之后，字符串的长度。

```
# 对非空字符串进行 SETRANGE

redis> SET greeting "hello world"
OK

redis> SETRANGE greeting 6 "Redis"
(integer) 11

redis> GET greeting
"hello Redis"

# 对空字符串/不存在的 key 进行 SETRANGE

redis> EXISTS empty_string
(integer) 0

redis> SETRANGE empty_string 5 "Redis!"   # 对不存在的 key 使用 SETRA
(integer) 11

redis> GET empty_string                    # 空白处被"\x00"填充
"\x00\x00\x00\x00\x00Redis!"
```

模式

因为有了 SETRANGE 和 *GETRANGE* 命令，你可以将 Redis 字符串用作具有O(1)随机访问时间的线性数组，这在很多真实用例中都是非常快速且高效的储存方式，具体请参考 *APPEND* 命令的『模式：时间序列』部分。

STRLEN

STRLEN key

返回 `key` 所储存的字符串值的长度。

当 `key` 储存的不是字符串值时，返回一个错误。

可用版本：

>= 2.2.0

复杂度：

O(1)

返回值：

字符串值的长度。当 `key` 不存在时，返回 `0` 。

```
# 获取字符串的长度

redis> SET mykey "Hello world"
OK

redis> STRLEN mykey
(integer) 11

# 不存在的 key 长度为 0

redis> STRLEN nonexisting
(integer) 0
```

Hash（哈希表）

HDEL

HDEL key field [field ...]

删除哈希表 `key` 中的一个或多个指定域，不存在的域将被忽略。

Note

在Redis2.4以下的版本里， HDEL 每次只能删除单个域，如果你需要在一个原子时间内删除多个域，请将命令包含在 *MULTI* / *EXEC* 块内。

可用版本：

>= 2.0.0

时间复杂度:

O(N)， `N` 为要删除的域的数量。

返回值:

被成功移除的域的数量，不包括被忽略的域。

```
# 测试数据

redis> HGETALL abbr
1) "a"
2) "apple"
3) "b"
4) "banana"
5) "c"
6) "cat"
7) "d"
8) "dog"

# 删除单个域

redis> HDEL abbr a
(integer) 1

# 删除不存在的域

redis> HDEL abbr not-exists-field
(integer) 0

# 删除多个域

redis> HDEL abbr b c
(integer) 2

redis> HGETALL abbr
1) "d"
2) "dog"
```

HEXISTS

HEXISTS key field

查看哈希表 `key` 中，给定域 `field` 是否存在。

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

如果哈希表含有给定域，返回 `1` 。如果哈希表不含有给定域，或 `key` 不存在，返回 `0` 。

```
redis> HEXISTS phone myphone
(integer) 0

redis> HSET phone myphone nokia-1110
(integer) 1

redis> HEXISTS phone myphone
(integer) 1
```

HGET

HGET key field

返回哈希表 `key` 中给定域 `field` 的值。

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

给定域的值。当给定域不存在或是给定 `key` 不存在时，返回 `nil` 。

```
# 域存在

redis> HSET site redis redis.com
(integer) 1

redis> HGET site redis
"redis.com"

# 域不存在

redis> HGET site mysql
(nil)
```

HGETALL

HGETALL key

返回哈希表 `key` 中，所有的域和值。

在返回值里，紧跟每个域名(field name)之后是域的值(value)，所以返回值的长度是哈希表大小的两倍。

可用版本：

>= 2.0.0

时间复杂度：

O(N)， `N` 为哈希表的大小。

返回值：

以列表形式返回哈希表的域和域的值。若 `key` 不存在，返回空列表。

```
redis> HSET people jack "Jack Sparrow"
(integer) 1

redis> HSET people gump "Forrest Gump"
(integer) 1

redis> HGETALL people
1) "jack"          # 域
2) "Jack Sparrow"  # 值
3) "gump"
4) "Forrest Gump"
```

HINCRBY

HINCRBY key field increment

为哈希表 `key` 中的域 `field` 的值加上增量 `increment` 。

增量也可以为负数，相当于对给定域进行减法操作。

如果 `key` 不存在，一个新的哈希表被创建并执行 HINCRBY 命令。

如果域 `field` 不存在，那么在执行命令前，域的值被初始化为 `0` 。

对一个储存字符串值的域 `field` 执行 HINCRBY 命令将造成一个错误。

本操作的值被限制在 64 位(bit)有符号数字表示之内。

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

执行 HINCRBY 命令之后，哈希表 `key` 中域 `field` 的值。

```
# increment 为正数

redis> HEXISTS counter page_view    # 对空域进行设置
(integer) 0

redis> HINCRBY counter page_view 200
(integer) 200

redis> HGET counter page_view
"200"

# increment 为负数

redis> HGET counter page_view
"200"

redis> HINCRBY counter page_view -50
(integer) 150

redis> HGET counter page_view
"150"

# 尝试对字符串值的域执行HINCRBY命令

redis> HSET myhash string hello,world       # 设定一个字符串值
(integer) 1

redis> HGET myhash string
"hello,world"

redis> HINCRBY myhash string 1              # 命令执行失败，错误。
(error) ERR hash value is not an integer

redis> HGET myhash string                   # 原值不变
"hello,world"
```

HINCRBYFLOAT

HINCRBYFLOAT key field increment

为哈希表 `key` 中的域 `field` 加上浮点数增量 `increment` 。

如果哈希表中没有域 `field` ，那么 HINCRBYFLOAT 会先将域 `field` 的值设为 `0` ，然后再执行加法操作。

如果键 `key` 不存在，那么 HINCRBYFLOAT 会先创建一个哈希表，再创建域 `field` ，最后再执行加法操作。

当以下任意一个条件发生时，返回一个错误：

- 域 `field` 的值不是字符串类型(因为 redis 中的数字和浮点数都以字符串的形式保存，所以它们都属于字符串类型）
- 域 `field` 当前的值或给定的增量 `increment` 不能解释(parse)为双精度浮点数(double precision floating point number)

HINCRBYFLOAT 命令的详细功能和 *INCRBYFLOAT* 命令类似，请查看 *INCRBYFLOAT* 命令获取更多相关信息。

可用版本：

>= 2.6.0

时间复杂度：

O(1)

返回值：

执行加法操作之后 `field` 域的值。

```
# 值和增量都是普通小数

redis> HSET mykey field 10.50
(integer) 1
redis> HINCRBYFLOAT mykey field 0.1
"10.6"

# 值和增量都是指数符号

redis> HSET mykey field 5.0e3
(integer) 0
redis> HINCRBYFLOAT mykey field 2.0e2
"5200"

# 对不存在的键执行 HINCRBYFLOAT

redis> EXISTS price
(integer) 0
redis> HINCRBYFLOAT price milk 3.5
"3.5"
redis> HGETALL price
1) "milk"
2) "3.5"

# 对不存在的域进行 HINCRBYFLOAT

redis> HGETALL price
1) "milk"
2) "3.5"
redis> HINCRBYFLOAT price coffee 4.5   # 新增 coffee 域
"4.5"
redis> HGETALL price
1) "milk"
2) "3.5"
3) "coffee"
4) "4.5"
```

HKEYS

HKEYS key

返回哈希表 `key` 中的所有域。

可用版本：

>= 2.0.0

时间复杂度：

O(N)， `N` 为哈希表的大小。

返回值：

一个包含哈希表中所有域的表。当 `key` 不存在时，返回一个空表。

```
# 哈希表非空

redis> HMSET website google www.google.com yahoo www.yahoo.com
OK

redis> HKEYS website
1) "google"
2) "yahoo"

# 空哈希表/key不存在

redis> EXISTS fake_key
(integer) 0

redis> HKEYS fake_key
(empty list or set)
```

HLEN

HLEN key

返回哈希表 `key` 中域的数量。

时间复杂度：

O(1)

返回值：

哈希表中域的数量。当 `key` 不存在时，返回 `0` 。

```
redis> HSET db redis redis.com
(integer) 1

redis> HSET db mysql mysql.com
(integer) 1

redis> HLEN db
(integer) 2

redis> HSET db mongodb mongodb.org
(integer) 1

redis> HLEN db
(integer) 3
```

HMGET

HMGET key field [field ...]

返回哈希表 `key` 中，一个或多个给定域的值。

如果给定的域不存在于哈希表，那么返回一个 `nil` 值。

因为不存在的 `key` 被当作一个空哈希表来处理，所以对一个不存在的 `key` 进行 HMGET 操作将返回一个只带有 `nil` 值的表。

可用版本：

>= 2.0.0

时间复杂度：

O(N)， `N` 为给定域的数量。

返回值：

一个包含多个给定域的关联值的表，表值的排列顺序和给定域参数的请求顺序一样。

```
redis> HMSET pet dog "doudou" cat "nounou"    # 一次设置多个域
OK

redis> HMGET pet dog cat fake_pet             # 返回值的顺序和传入参数
1) "doudou"
2) "nounou"
3) (nil)                                      # 不存在的域返回nil值
```

HMSET

HMSET key field value [field value ...]

同时将多个 `field-value` (域-值)对设置到哈希表 `key` 中。

此命令会覆盖哈希表中已存在的域。

如果 `key` 不存在，一个空哈希表被创建并执行 HMSET 操作。

可用版本：

>= 2.0.0

时间复杂度：

O(N)， `N` 为 `field-value` 对的数量。

返回值：

如果命令执行成功，返回 `OK` 。当 `key` 不是哈希表(hash)类型时，返回一个错误。

```
redis> HMSET website google www.google.com yahoo www.yahoo.com
OK

redis> HGET website google
"www.google.com"

redis> HGET website yahoo
"www.yahoo.com"
```

HSET

HSET key field value

将哈希表 `key` 中的域 `field` 的值设为 `value` 。

如果 `key` 不存在，一个新的哈希表被创建并进行 HSET 操作。

如果域 `field` 已经存在于哈希表中，旧值将被覆盖。

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

如果 `field` 是哈希表中的一个新建域，并且值设置成功，返回 `1` 。如果哈希表中域 `field` 已经存在且旧值已被新值覆盖，返回 `0` 。

```
redis> HSET website google "www.g.cn"       # 设置一个新域
(integer) 1

redis> HSET website google "www.google.com" # 覆盖一个旧域
(integer) 0
```

HSETNX

HSETNX key field value

将哈希表 `key` 中的域 `field` 的值设置为 `value` ，当且仅当域 `field` 不存在。

若域 `field` 已经存在，该操作无效。

如果 `key` 不存在，一个新哈希表被创建并执行 HSETNX 命令。

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

设置成功，返回 `1` 。如果给定域已经存在且没有操作被执行，返回 `0` 。

```
redis> HSETNX nosql key-value-store redis
(integer) 1

redis> HSETNX nosql key-value-store redis       # 操作无效， 域 key-va
(integer) 0
```

HVALS

HVALS key

返回哈希表 `key` 中所有域的值。

可用版本：

>= 2.0.0

时间复杂度：

O(N)， `N` 为哈希表的大小。

返回值：

一个包含哈希表中所有值的表。当 `key` 不存在时，返回一个空表。

```
# 非空哈希表

redis> HMSET website google www.google.com yahoo www.yahoo.com
OK

redis> HVALS website
1) "www.google.com"
2) "www.yahoo.com"

# 空哈希表/不存在的key

redis> EXISTS not_exists
(integer) 0

redis> HVALS not_exists
(empty list or set)
```

HSCAN

HSCAN key cursor [MATCH pattern] [COUNT count]

具体信息请参考 *SCAN* 命令。

List（列表）

BLPOP

BLPOP key [key ...] timeout

BLPOP 是列表的阻塞式(blocking)弹出原语。

它是 *LPOP* 命令的阻塞版本，当给定列表内没有任何元素可供弹出的时候，连接将被 BLPOP 命令阻塞，直到等待超时或发现可弹出元素为止。

当给定多个 `key` 参数时，按参数 `key` 的先后顺序依次检查各个列表，弹出第一个非空列表的头元素。

非阻塞行为

当 BLPOP 被调用时，如果给定 `key` 内至少有一个非空列表，那么弹出遇到的第一个非空列表的头元素，并和被弹出元素所属的列表的名字一起，组成结果返回给调用者。

当存在多个给定 `key` 时， BLPOP 按给定 `key` 参数排列的先后顺序，依次检查各个列表。

假设现在有 `job` 、 `command` 和 `request` 三个列表，其中 `job` 不存在， `command` 和 `request` 都持有非空列表。考虑以下命令：

`BLPOP job command request 0`

BLPOP 保证返回的元素来自 `command` ，因为它是按”查找 `job` -> 查找 `command` -> 查找 `request` “这样的顺序，第一个找到的非空列表。

```
redis> DEL job command request           # 确保key都被删除
(integer) 0

redis> LPUSH command "update system..."  # 为command列表增加一个值
(integer) 1

redis> LPUSH request "visit page"        # 为request列表增加一个值
(integer) 1

redis> BLPOP job command request 0       # job 列表为空，被跳过，紧接着
1) "command"                             # 弹出元素所属的列表
2) "update system..."                    # 弹出元素所属的值
```

阻塞行为

如果所有给定 `key` 都不存在或包含空列表，那么 BLPOP 命令将阻塞连接，直到等待超时，或有另一个客户端对给定 `key` 的任意一个执行 *LPUSH* 或 *RPUSH* 命令为止。

超时参数 `timeout` 接受一个以秒为单位的数字作为值。超时参数设为 `0` 表示阻塞时间可以无限期延长(block indefinitely) 。

```
redis> EXISTS job                # 确保两个 key 都不存在
(integer) 0
redis> EXISTS command
(integer) 0

redis> BLPOP job command 300     # 因为key一开始不存在，所以操作会被阻塞
1) "job"                         # 这里被 push 的是 job
2) "do my home work"             # 被弹出的值
(26.26s)                         # 等待的秒数

redis> BLPOP job command 5       # 等待超时的情况
(nil)
(5.66s)                          # 等待的秒数
```

相同的**key**被多个客户端同时阻塞

相同的 `key` 可以被多个客户端同时阻塞。

不同的客户端被放进一个队列中，按『先阻塞先服务』(first-BLPOP，first-served)的顺序为 `key` 执行 BLPOP 命令。

在**MULTI/EXEC**事务中的**BLPOP**

BLPOP 可以用于流水线(pipline,批量地发送多个命令并读入多个回复)，但把它用在 *MULTI* / *EXEC* 块当中没有意义。因为这要求整个服务器被阻塞以保证块执行时的原子性，该行为阻止了其他客户端执行 *LPUSH* 或 *RPUSH* 命令。

因此，一个被包裹在 *MULTI* / *EXEC* 块内的 BLPOP 命令，行为表现得就像 *LPOP* 一样，对空列表返回 `nil` ，对非空列表弹出列表元素，不进行任何阻塞操作。

```
# 对非空列表进行操作

redis> RPUSH job programming
(integer) 1

redis> MULTI
OK

redis> BLPOP job 30
QUEUED

redis> EXEC           # 不阻塞，立即返回
1) 1) "job"
   2) "programming"

# 对空列表进行操作

redis> LLEN job      # 空列表
(integer) 0

redis> MULTI
OK

redis> BLPOP job 30
QUEUED

redis> EXEC          # 不阻塞，立即返回
1) (nil)
```

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

如果列表为空，返回一个 `nil` 。否则，返回一个含有两个元素的列表，第一个元素是被弹出元素所属的 `key` ，第二个元素是被弹出元素的值。

模式：事件提醒

有时候，为了等待一个新元素到达数据中，需要使用轮询的方式对数据进行探查。

另一种更好的方式是，使用系统提供的阻塞原语，在新元素到达时立即进行处理，而新元素还没到达时，就一直阻塞住，避免轮询占用资源。

对于 Redis， 我们似乎需要一个阻塞版的 *SPOP* 命令，但实际上，使用 BLPOP 或者 *BRPOP* 就能很好地解决这个问题。

使用元素的客户端(消费者)可以执行类似以下的代码：

```
LOOP forever
    WHILE SPOP(key) returns elements
        ... process elements ...
    END
    BRPOP helper_key
END
```

添加元素的客户端(消费者)则执行以下代码：

```
MULTI
    SADD key element
    LPUSH helper_key x
EXEC
```

BRPOP

BRPOP key [key ...] timeout

BRPOP 是列表的阻塞式(blocking)弹出原语。

它是 *RPOP* 命令的阻塞版本，当给定列表内没有任何元素可供弹出的时候，连接将被 BRPOP 命令阻塞，直到等待超时或发现可弹出元素为止。

当给定多个 `key` 参数时，按参数 `key` 的先后顺序依次检查各个列表，弹出第一个非空列表的尾部元素。

关于阻塞操作的更多信息，请查看 *BLPOP* 命令， BRPOP 除了弹出元素的位置和 *BLPOP* 不同之外，其他表现一致。

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

假如在指定时间内没有任何元素被弹出，则返回一个 `nil` 和等待时长。反之，返回一个含有两个元素的列表，第一个元素是被弹出元素所属的 `key` ，第二个元素是被弹出元素的值。

```
redis> LLEN course
(integer) 0

redis> RPUSH course algorithm001
(integer) 1

redis> RPUSH course c++101
(integer) 2

redis> BRPOP course 30
1) "course"             # 被弹出元素所属的列表键
2) "c++101"             # 被弹出的元素
```

BRPOPLPUSH

BRPOPLPUSH source destination timeout

BRPOPLPUSH 是 *RPOPLPUSH* 的阻塞版本，当给定列表 `source` 不为空时，BRPOPLPUSH 的表现和 *RPOPLPUSH* 一样。

当列表 `source` 为空时， BRPOPLPUSH 命令将阻塞连接，直到等待超时，或有另一个客户端对 `source` 执行 *LPUSH* 或 *RPUSH* 命令为止。

超时参数 `timeout` 接受一个以秒为单位的数字作为值。超时参数设为 `0` 表示阻塞时间可以无限期延长(block indefinitely) 。

更多相关信息，请参考 *RPOPLPUSH* 命令。

可用版本：

>= 2.2.0

时间复杂度：

O(1)

返回值：

假如在指定时间内没有任何元素被弹出，则返回一个 `nil` 和等待时长。反之，返回一个含有两个元素的列表，第一个元素是被弹出元素的值，第二个元素是等待时长。

```
# 非空列表

redis> BRPOPLPUSH msg reciver 500
"hello moto"                        # 弹出元素的值
(3.38s)                             # 等待时长

redis> LLEN reciver
(integer) 1

redis> LRANGE reciver 0 0
1) "hello moto"

# 空列表

redis> BRPOPLPUSH msg reciver 1
(nil)
(1.34s)
```

模式：安全队列

参考 *RPOPLPUSH* 命令的『安全队列』模式。

模式：循环列表

参考 *RPOPLPUSH* 命令的『循环列表』模式。

LINDEX

LINDEX key index

返回列表 `key` 中，下标为 `index` 的元素。

下标(index)参数 `start` 和 `stop` 都以 `0` 为底，也就是说，以 `0` 表示列表的第一个元素，以 `1` 表示列表的第二个元素，以此类推。

你也可以使用负数下标，以 `-1` 表示列表的最后一个元素， `-2` 表示列表的倒数第二个元素，以此类推。

如果 `key` 不是列表类型，返回一个错误。

可用版本：

>= 1.0.0

时间复杂度：

O(N)， `N` 为到达下标 `index` 过程中经过的元素数量。因此，对列表的头元素和尾元素执行 LINDEX 命令，复杂度为O(1)。

返回值:

列表中下标为 `index` 的元素。如果 `index` 参数的值不在列表的区间范围内(out of range)，返回 `nil` 。

```
redis> LPUSH mylist "World"
(integer) 1

redis> LPUSH mylist "Hello"
(integer) 2

redis> LINDEX mylist 0
"Hello"

redis> LINDEX mylist -1
"World"

redis> LINDEX mylist 3        # index不在 mylist 的区间范围内
(nil)
```

LINSERT

LINSERT key BEFORE|AFTER pivot value

将值 `value` 插入到列表 `key` 当中，位于值 `pivot` 之前或之后。

当 `pivot` 不存在于列表 `key` 时，不执行任何操作。

当 `key` 不存在时， `key` 被视为空列表，不执行任何操作。

如果 `key` 不是列表类型，返回一个错误。

可用版本：

>= 2.2.0

时间复杂度:

O(N)， `N` 为寻找 `pivot` 过程中经过的元素数量。

返回值:

如果命令执行成功，返回插入操作完成之后，列表的长度。如果没有找到 `pivot` ，返回 `-1` 。如果 `key` 不存在或为空列表，返回 `0` 。

```
redis> RPUSH mylist "Hello"
(integer) 1

redis> RPUSH mylist "World"
(integer) 2

redis> LINSERT mylist BEFORE "World" "There"
(integer) 3

redis> LRANGE mylist 0 -1
1) "Hello"
2) "There"
3) "World"

# 对一个非空列表插入，查找一个不存在的 pivot

redis> LINSERT mylist BEFORE "go" "let's"
(integer) -1                                    # 失败

# 对一个空列表执行 LINSERT 命令

redis> EXISTS fake_list
(integer) 0

redis> LINSERT fake_list BEFORE "nono" "gogogog"
(integer) 0                                     # 失败
```

LLEN

LLEN key

返回列表 `key` 的长度。

如果 `key` 不存在，则 `key` 被解释为一个空列表，返回 `0` .

如果 `key` 不是列表类型，返回一个错误。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

列表 `key` 的长度。

```
# 空列表

redis> LLEN job
(integer) 0

# 非空列表

redis> LPUSH job "cook food"
(integer) 1

redis> LPUSH job "have lunch"
(integer) 2

redis> LLEN job
(integer) 2
```

LPOP

LPOP key

移除并返回列表 `key` 的头元素。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

列表的头元素。当 `key` 不存在时，返回 `nil` 。

```
redis> LLEN course
(integer) 0

redis> RPUSH course algorithm001
(integer) 1

redis> RPUSH course c++101
(integer) 2

redis> LPOP course  # 移除头元素
"algorithm001"
```

LPUSH

LPUSH key value [value ...]

将一个或多个值 `value` 插入到列表 `key` 的表头

如果有多个 `value` 值，那么各个 `value` 值按从左到右的顺序依次插入到表头： 比如说，对空列表 `mylist` 执行命令 `LPUSH mylist a b c` ，列表的值将是 `c b a` ，这等同于原子性地执行 `LPUSH mylist a` 、 `LPUSH mylist b` 和 `LPUSH mylist c` 三个命令。

如果 `key` 不存在，一个空列表会被创建并执行 LPUSH 操作。

当 `key` 存在但不是列表类型时，返回一个错误。

Note

在Redis 2.4版本以前的 LPUSH 命令，都只接受单个 `value` 值。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

执行 LPUSH 命令后，列表的长度。

```
# 加入单个元素

redis> LPUSH languages python
(integer) 1

# 加入重复元素

redis> LPUSH languages python
(integer) 2

redis> LRANGE languages 0 -1     # 列表允许重复元素
1) "python"
2) "python"

# 加入多个元素

redis> LPUSH mylist a b c
(integer) 3

redis> LRANGE mylist 0 -1
1) "c"
2) "b"
3) "a"
```

LRANGE

LRANGE key start stop

返回列表 `key` 中指定区间内的元素，区间以偏移量 `start` 和 `stop` 指定。

下标(index)参数 `start` 和 `stop` 都以 `0` 为底，也就是说，以 `0` 表示列表的第一个元素，以 `1` 表示列表的第二个元素，以此类推。

你也可以使用负数下标，以 `-1` 表示列表的最后一个元素， `-2` 表示列表的倒数第二个元素，以此类推。

注意**LRANGE**命令和编程语言区间函数的区别

假如你有一个包含一百个元素的列表，对该列表执行 `LRANGE list 0 10` ，结果是一个包含11个元素的列表，这表明 `stop` 下标也在 LRANGE 命令的取值范围之内(闭区间)，这和某些语言的区间函数可能不一致，比如Ruby的 `Range.new` 、 `Array#slice` 和Python的 `range()` 函数。

超出范围的下标

超出范围的下标值不会引起错误。

如果 `start` 下标比列表的最大下标 `end` (`LLEN list` 减去 `1`)还要大，那么 LRANGE 返回一个空列表。

如果 `stop` 下标比 `end` 下标还要大，Redis将 `stop` 的值设置为 `end` 。

可用版本：

>= 1.0.0

时间复杂度:

O(S+N)， `S` 为偏移量 `start` ， `N` 为指定区间内元素的数量。

返回值:

一个列表，包含指定区间内的元素。

```
redis> RPUSH fp-language lisp
(integer) 1

redis> LRANGE fp-language 0 0
1) "lisp"

redis> RPUSH fp-language scheme
(integer) 2

redis> LRANGE fp-language 0 1
1) "lisp"
2) "scheme"
```

LREM

LREM key count value

根据参数 `count` 的值，移除列表中与参数 `value` 相等的元素。

`count` 的值可以是以下几种：

- `count > 0` : 从表头开始向表尾搜索，移除与 `value` 相等的元素，数量为 `count` 。
- `count < 0` : 从表尾开始向表头搜索，移除与 `value` 相等的元素，数量为 `count` 的绝对值。
- `count = 0` : 移除表中所有与 `value` 相等的值。

可用版本：

>= 1.0.0

时间复杂度：

O(N)， `N` 为列表的长度。

返回值：

被移除元素的数量。因为不存在的 `key` 被视作空表(empty list)，所以当 `key` 不存在时， LREM 命令总是返回 `0` 。

```
# 先创建一个表，内容排列是
# morning hello morning helllo morning

redis> LPUSH greet "morning"
(integer) 1
redis> LPUSH greet "hello"
(integer) 2
redis> LPUSH greet "morning"
(integer) 3
redis> LPUSH greet "hello"
(integer) 4
redis> LPUSH greet "morning"
(integer) 5

redis> LRANGE greet 0 4         # 查看所有元素
1) "morning"
2) "hello"
3) "morning"
4) "hello"
5) "morning"

redis> LREM greet 2 morning     # 移除从表头到表尾，最先发现的两个 morni
(integer) 2                     # 两个元素被移除

redis> LLEN greet               # 还剩 3 个元素
(integer) 3

redis> LRANGE greet 0 2
1) "hello"
2) "hello"
3) "morning"

redis> LREM greet -1 morning    # 移除从表尾到表头，第一个 morning
(integer) 1

redis> LLEN greet               # 剩下两个元素
(integer) 2

redis> LRANGE greet 0 1
1) "hello"
2) "hello"

redis> LREM greet 0 hello      # 移除表中所有 hello
(integer) 2                    # 两个 hello 被移除

redis> LLEN greet
(integer) 0
```

LSET

LSET key index value

将列表 `key` 下标为 `index` 的元素的值设置为 `value` 。

当 `index` 参数超出范围，或对一个空列表(`key` 不存在)进行 LSET 时，返回一个错误。

关于列表下标的更多信息，请参考 *LINDEX* 命令。

可用版本：

>= 1.0.0

时间复杂度：

对头元素或尾元素进行 LSET 操作，复杂度为 O(1)。其他情况下，为 O(N)， `N` 为列表的长度。

返回值：

操作成功返回 `ok` ，否则返回错误信息。

```
# 对空列表(key 不存在)进行 LSET

redis> EXISTS list
(integer) 0

redis> LSET list 0 item
(error) ERR no such key

# 对非空列表进行 LSET

redis> LPUSH job "cook food"
(integer) 1

redis> LRANGE job 0 0
1) "cook food"

redis> LSET job 0 "play game"
OK

redis> LRANGE job  0 0
1) "play game"

# index 超出范围

redis> LLEN list                    # 列表长度为 1
(integer) 1

redis> LSET list 3 'out of range'
(error) ERR index out of range
```

LTRIM

LTRIM key start stop

对一个列表进行修剪(trim)，就是说，让列表只保留指定区间内的元素，不在指定区间之内的元素都将被删除。

举个例子，执行命令 `LTRIM list 0 2` ，表示只保留列表 `list` 的前三个元素，其余元素全部删除。

下标(index)参数 `start` 和 `stop` 都以 `0` 为底，也就是说，以 `0` 表示列表的第一个元素，以 `1` 表示列表的第二个元素，以此类推。

你也可以使用负数下标，以 `-1` 表示列表的最后一个元素， `-2` 表示列表的倒数第二个元素，以此类推。

当 `key` 不是列表类型时，返回一个错误。

LTRIM 命令通常和 *LPUSH* 命令或 *RPUSH* 命令配合使用，举个例子：

```
LPUSH log newest_log
LTRIM log 0 99
```

这个例子模拟了一个日志程序，每次将最新日志 `newest_log` 放到 `log` 列表中，并且只保留最新的 `100` 项。注意当这样使用 `LTRIM` 命令时，时间复杂度是O(1)，因为平均情况下，每次只有一个元素被移除。

注意**LTRIM**命令和编程语言区间函数的区别

假如你有一个包含一百个元素的列表 `list` ，对该列表执行 `LTRIM list 0 10` ，结果是一个包含11个元素的列表，这表明 `stop` 下标也在 LTRIM 命令的取值范围之内(闭区间)，这和某些语言的区间函数可能不一致，比如Ruby的 `Range.new` 、 `Array#slice` 和Python的 `range()` 函数。

超出范围的下标

超出范围的下标值不会引起错误。

如果 `start` 下标比列表的最大下标 `end` (`LLEN list` 减去 `1`)还要大，或者 `start > stop` ， LTRIM 返回一个空列表(因为 LTRIM 已经将整个列表清空)。

如果 `stop` 下标比 `end` 下标还要大，Redis将 `stop` 的值设置为 `end` 。

可用版本：

>= 1.0.0

时间复杂度:

O(N)， `N` 为被移除的元素的数量。

返回值:

命令执行成功时，返回 `ok` 。

```
# 情况 1： 常见情况， start 和 stop 都在列表的索引范围之内

redis> LRANGE alpha 0 -1       # alpha 是一个包含 5 个字符串的列表
1) "h"
2) "e"
3) "l"
4) "l"
5) "o"

redis> LTRIM alpha 1 -1        # 删除 alpha 列表索引为 0 的元素
OK

redis> LRANGE alpha 0 -1       # "h" 被删除了
1) "e"
2) "l"
3) "l"
4) "o"

# 情况 2： stop 比列表的最大下标还要大

redis> LTRIM alpha 1 10086     # 保留 alpha 列表索引 1 至索引 10086 上
OK

redis> LRANGE alpha 0 -1       # 只有索引 0 上的元素 "e" 被删除了，其他
1) "l"
2) "l"
3) "o"

# 情况 3： start 和 stop 都比列表的最大下标要大，并且 start < stop

redis> LTRIM alpha 10086 123321
OK

redis> LRANGE alpha 0 -1        # 列表被清空
(empty list or set)

# 情况 4： start 和 stop 都比列表的最大下标要大，并且 start > stop

redis> RPUSH new-alpha "h" "e" "l" "l" "o"     # 重新建立一个新列表
(integer) 5

redis> LRANGE new-alpha 0 -1
1) "h"
2) "e"
3) "l"
```

```
4) "l"
5) "o"

redis> LTRIM new-alpha 123321 10086    # 执行 LTRIM
OK

redis> LRANGE new-alpha 0 -1           # 同样被清空
(empty list or set)
```

RPOP

RPOP key

移除并返回列表 `key` 的尾元素。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

列表的尾元素。当 `key` 不存在时，返回 `nil` 。

```
redis> RPUSH mylist "one"
(integer) 1

redis> RPUSH mylist "two"
(integer) 2

redis> RPUSH mylist "three"
(integer) 3

redis> RPOP mylist           # 返回被弹出的元素
"three"

redis> LRANGE mylist 0 -1    # 列表剩下的元素
1) "one"
2) "two"
```

RPOPLPUSH

RPOPLPUSH source destination

命令 RPOPLPUSH 在一个原子时间内，执行以下两个动作：

- 将列表 `source` 中的最后一个元素(尾元素)弹出，并返回给客户端。
- 将 `source` 弹出的元素插入到列表 `destination` ，作为 `destination` 列表的的头元素。

举个例子，你有两个列表 `source` 和 `destination` ， `source` 列表有元素 `a, b, c` ， `destination` 列表有元素 `x, y, z` ，执行 `RPOPLPUSH source destination` 之后， `source` 列表包含元素 `a, b` ， `destination` 列表包含元素 `c, x, y, z` ，并且元素 `c` 会被返回给客户端。

如果 `source` 不存在，值 `nil` 被返回，并且不执行其他动作。

如果 `source` 和 `destination` 相同，则列表中的表尾元素被移动到表头，并返回该元素，可以把这种特殊情况视作列表的旋转(rotation)操作。

可用版本：

>= 1.2.0

时间复杂度：

O(1)

返回值：

被弹出的元素。

```
# source 和 destination 不同

redis> LRANGE alpha 0 -1         # 查看所有元素
1) "a"
2) "b"
3) "c"
4) "d"

redis> RPOPLPUSH alpha reciver   # 执行一次 RPOPLPUSH 看看
"d"

redis> LRANGE alpha 0 -1
1) "a"
2) "b"
3) "c"

redis> LRANGE reciver 0 -1
1) "d"
```

```
redis> RPOPLPUSH alpha reciver   # 再执行一次，证实 RPOP 和 LPUSH 的位
"c"

redis> LRANGE alpha 0 -1
1) "a"
2) "b"

redis> LRANGE reciver 0 -1
1) "c"
2) "d"

# source 和 destination 相同

redis> LRANGE number 0 -1
1) "1"
2) "2"
3) "3"
4) "4"

redis> RPOPLPUSH number number
"4"

redis> LRANGE number 0 -1           # 4 被旋转到了表头
1) "4"
2) "1"
3) "2"
4) "3"

redis> RPOPLPUSH number number
"3"

redis> LRANGE number 0 -1           # 这次是 3 被旋转到了表头
1) "3"
2) "4"
3) "1"
4) "2"
```

模式： 安全的队列

Redis的列表经常被用作队列(queue)，用于在不同程序之间有序地交换消息(message)。一个客户端通过 *LPUSH* 命令将消息放入队列中，而另一个客户端通过 *RPOP* 或者 *BRPOP* 命令取出队列中等待时间最长的消息。

不幸的是，上面的队列方法是『不安全』的，因为在这个过程中，一个客户端可能在取出一个消息之后崩溃，而未处理完的消息也就因此丢失。

使用 RPOPLPUSH 命令(或者它的阻塞版本 *BRPOPLPUSH*)可以解决这个问题：因为它不仅返回一个消息，同时还将这个消息添加到另一个备份列表当中，如果一切正常的话，当一个客户端完成某个消息的处理之后，可以用 *LREM* 命令将这个消息从备份表删除。

最后，还可以添加一个客户端专门用于监视备份表，它自动地将超过一定处理时限的消息重新放入队列中去(负责处理该消息的客户端可能已经崩溃)，这样就不会丢失任何消息了。

模式：循环列表

通过使用相同的 `key` 作为 RPOPLPUSH 命令的两个参数，客户端可以用一个接一个地获取列表元素的方式，取得列表的所有元素，而不必像 *LRANGE* 命令那样一下子将所有列表元素都从服务器传送到客户端中(两种方式的总复杂度都是 O(N))。

以上的模式甚至在以下的两个情况下也能正常工作：

- 有多个客户端同时对同一个列表进行旋转(rotating)，它们获取不同的元素，直到所有元素都被读取完，之后又从头开始。
- 有客户端在向列表尾部(右边)添加新元素。

这个模式使得我们可以很容易实现这样一类系统：有 N 个客户端，需要连续不断地对一些元素进行处理，而且处理的过程必须尽可能地快。一个典型的例子就是服务器的监控程序：它们需要在尽可能短的时间内，并行地检查一组网站，确保它们的可访问性。

注意，使用这个模式的客户端是易于扩展(scala)且安全(reliable)的，因为就算接收到元素的客户端失败，元素还是保存在列表里面，不会丢失，等到下个迭代来临的时候，别的客户端又可以继续处理这些元素了。

RPUSH

RPUSH key value [value ...]

将一个或多个值 `value` 插入到列表 `key` 的表尾(最右边)。

如果有多个 `value` 值，那么各个 `value` 值按从左到右的顺序依次插入到表尾：比如对一个空列表 `mylist` 执行 `RPUSH mylist a b c` ，得出的结果列表为 `a b c` ，等同于执行命令 `RPUSH mylist a` 、 `RPUSH mylist b` 、 `RPUSH mylist c` 。

如果 `key` 不存在，一个空列表会被创建并执行 RPUSH 操作。

当 `key` 存在但不是列表类型时，返回一个错误。

Note

在 Redis 2.4 版本以前的 RPUSH 命令，都只接受单个 `value` 值。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

执行 RPUSH 操作后，表的长度。

```
# 添加单个元素

redis> RPUSH languages c
(integer) 1

# 添加重复元素

redis> RPUSH languages c
(integer) 2

redis> LRANGE languages 0 -1 # 列表允许重复元素
1) "c"
2) "c"

# 添加多个元素

redis> RPUSH mylist a b c
(integer) 3

redis> LRANGE mylist 0 -1
1) "a"
2) "b"
3) "c"
```

RPUSHX

RPUSHX key value

将值 `value` 插入到列表 `key` 的表尾，当且仅当 `key` 存在并且是一个列表。

和 *RPUSH* 命令相反，当 `key` 不存在时， RPUSHX 命令什么也不做。

可用版本：

>= 2.2.0

时间复杂度：

O(1)

返回值：

RPUSHX 命令执行之后，表的长度。

```
# key不存在

redis> LLEN greet
(integer) 0

redis> RPUSHX greet "hello"     # 对不存在的 key 进行 RPUSHX，PUSH 失
(integer) 0

# key 存在且是一个非空列表

redis> RPUSH greet "hi"         # 先用 RPUSH 插入一个元素
(integer) 1

redis> RPUSHX greet "hello"     # greet 现在是一个列表类型，RPUSHX 操作
(integer) 2

redis> LRANGE greet 0 -1
1) "hi"
2) "hello"
```

Set（集合）

SADD

SADD key member [member ...]

将一个或多个 `member` 元素加入到集合 `key` 当中，已经存在于集合的 `member` 元素将被忽略。

假如 `key` 不存在，则创建一个只包含 `member` 元素作成员的集合。

当 `key` 不是集合类型时，返回一个错误。

Note

在Redis2.4版本以前， SADD 只接受单个 `member` 值。

可用版本：

>= 1.0.0

时间复杂度:

O(N)， `N` 是被添加的元素的数量。

返回值:

被添加到集合中的新元素的数量，不包括被忽略的元素。

```
# 添加单个元素

redis> SADD bbs "discuz.net"
(integer) 1

# 添加重复元素

redis> SADD bbs "discuz.net"
(integer) 0

# 添加多个元素

redis> SADD bbs "tianya.cn" "groups.google.com"
(integer) 2

redis> SMEMBERS bbs
1) "discuz.net"
2) "groups.google.com"
3) "tianya.cn"
```

SCARD

SCARD key

返回集合 `key` 的基数(集合中元素的数量)。

可用版本：

>= 1.0.0

时间复杂度:

O(1)

返回值：

集合的基数。当 `key` 不存在时，返回 `0` 。

```
redis> SADD tool pc printer phone
(integer) 3

redis> SCARD tool   # 非空集合
(integer) 3

redis> DEL tool
(integer) 1

redis> SCARD tool   # 空集合
(integer) 0
```

SDIFF

SDIFF key [key ...]

返回一个集合的全部成员，该集合是所有给定集合之间的差集。

不存在的 `key` 被视为空集。

可用版本：

>= 1.0.0

时间复杂度:

O(N)， `N` 是所有给定集合的成员数量之和。

返回值:

一个包含差集成员的列表。

```
redis> SMEMBERS peter's_movies
1) "bet man"
2) "start war"
3) "2012"

redis> SMEMBERS joe's_movies
1) "hi, lady"
2) "Fast Five"
3) "2012"

redis> SDIFF peter's_movies joe's_movies
1) "bet man"
2) "start war"
```

SDIFFSTORE

SDIFFSTORE destination key [key ...]

这个命令的作用和 *SDIFF* 类似，但它将结果保存到 `destination` 集合，而不是简单地返回结果集。

如果 `destination` 集合已经存在，则将其覆盖。

`destination` 可以是 `key` 本身。

可用版本：

>= 1.0.0

时间复杂度:

O(N)， `N` 是所有给定集合的成员数量之和。

返回值:

结果集中的元素数量。

```
redis> SMEMBERS joe's_movies
1) "hi, lady"
2) "Fast Five"
3) "2012"

redis> SMEMBERS peter's_movies
1) "bet man"
2) "start war"
3) "2012"

redis> SDIFFSTORE joe_diff_peter joe's_movies peter's_movies
(integer) 2

redis> SMEMBERS joe_diff_peter
1) "hi, lady"
2) "Fast Five"
```

SINTER

SINTER key [key ...]

返回一个集合的全部成员，该集合是所有给定集合的交集。

不存在的 `key` 被视为空集。

当给定集合当中有一个空集时，结果也为空集(根据集合运算定律)。

可用版本：

>= 1.0.0

时间复杂度:

O(N * M)， `N` 为给定集合当中基数最小的集合， `M` 为给定集合的个数。

返回值:

交集成员的列表。

```
redis> SMEMBERS group_1
1) "LI LEI"
2) "TOM"
3) "JACK"

redis> SMEMBERS group_2
1) "HAN MEIMEI"
2) "JACK"

redis> SINTER group_1 group_2
1) "JACK"
```

SINTER

SINTER key [key ...]

返回一个集合的全部成员，该集合是所有给定集合的交集。

不存在的 `key` 被视为空集。

当给定集合当中有一个空集时，结果也为空集(根据集合运算定律)。

可用版本：

>= 1.0.0

时间复杂度:

O(N * M)， `N` 为给定集合当中基数最小的集合， `M` 为给定集合的个数。

返回值:

交集成员的列表。

```
redis> SMEMBERS group_1
1) "LI LEI"
2) "TOM"
3) "JACK"

redis> SMEMBERS group_2
1) "HAN MEIMEI"
2) "JACK"

redis> SINTER group_1 group_2
1) "JACK"
```

SINTERSTORE

SINTERSTORE destination key [key ...]

这个命令类似于 *SINTER* 命令，但它将结果保存到 `destination` 集合，而不是简单地返回结果集。

如果 `destination` 集合已经存在，则将其覆盖。

`destination` 可以是 `key` 本身。

可用版本：

>= 1.0.0

时间复杂度:

O(N * M)， `N` 为给定集合当中基数最小的集合， `M` 为给定集合的个数。

返回值:

结果集中的成员数量。

```
redis> SMEMBERS songs
1) "good bye joe"
2) "hello,peter"

redis> SMEMBERS my_songs
1) "good bye joe"
2) "falling"

redis> SINTERSTORE song_interset songs my_songs
(integer) 1

redis> SMEMBERS song_interset
1) "good bye joe"
```

SISMEMBER

SISMEMBER key member

判断 `member` 元素是否集合 `key` 的成员。

可用版本：

>= 1.0.0

时间复杂度:

O(1)

返回值:

如果 `member` 元素是集合的成员，返回 `1` 。如果 `member` 元素不是集合的成员，或 `key` 不存在，返回 `0` 。

```
redis> SMEMBERS joe's_movies
1) "hi, lady"
2) "Fast Five"
3) "2012"

redis> SISMEMBER joe's_movies "bet man"
(integer) 0

redis> SISMEMBER joe's_movies "Fast Five"
(integer) 1
```

SMEMBERS

SMEMBERS key

返回集合 `key` 中的所有成员。

不存在的 `key` 被视为空集合。

可用版本：

>= 1.0.0

时间复杂度:

O(N)， `N` 为集合的基数。

返回值:

集合中的所有成员。

```
# key 不存在或集合为空

redis> EXISTS not_exists_key
(integer) 0

redis> SMEMBERS not_exists_key
(empty list or set)

# 非空集合

redis> SADD language Ruby Python Clojure
(integer) 3

redis> SMEMBERS language
1) "Python"
2) "Ruby"
3) "Clojure"
```

SMOVE

SMOVE source destination member

将 `member` 元素从 `source` 集合移动到 `destination` 集合。

SMOVE 是原子性操作。

如果 `source` 集合不存在或不包含指定的 `member` 元素，则 SMOVE 命令不执行任何操作，仅返回 `0` 。否则， `member` 元素从 `source` 集合中被移除，并添加到 `destination` 集合中去。

当 `destination` 集合已经包含 `member` 元素时， SMOVE 命令只是简单地将 `source` 集合中的 `member` 元素删除。

当 `source` 或 `destination` 不是集合类型时，返回一个错误。

可用版本：

>= 1.0.0

时间复杂度:

O(1)

返回值:

如果 `member` 元素被成功移除，返回 `1` 。如果 `member` 元素不是 `source` 集合的成员，并且没有任何操作对 `destination` 集合执行，那么返回 `0` 。

```
redis> SMEMBERS songs
1) "Billie Jean"
2) "Believe Me"

redis> SMEMBERS my_songs
(empty list or set)

redis> SMOVE songs my_songs "Believe Me"
(integer) 1

redis> SMEMBERS songs
1) "Billie Jean"

redis> SMEMBERS my_songs
1) "Believe Me"
```

SPOP

SPOP key

移除并返回集合中的一个随机元素。

如果只想获取一个随机元素，但不想该元素从集合中被移除的话，可以使用 *SRANDMEMBER* 命令。

可用版本：

>= 1.0.0

时间复杂度:

O(1)

返回值:

被移除的随机元素。当 `key` 不存在或 `key` 是空集时，返回 `nil` 。

```
redis> SMEMBERS db
1) "MySQL"
2) "MongoDB"
3) "Redis"

redis> SPOP db
"Redis"

redis> SMEMBERS db
1) "MySQL"
2) "MongoDB"

redis> SPOP db
"MySQL"

redis> SMEMBERS db
1) "MongoDB"
```

SRANDMEMBER

SRANDMEMBER key [count]

如果命令执行时，只提供了 `key` 参数，那么返回集合中的一个随机元素。

从 Redis 2.6 版本开始， SRANDMEMBER 命令接受可选的 `count` 参数：

- 如果 `count` 为正数，且小于集合基数，那么命令返回一个包含 `count` 个元素的数组，数组中的元素各不相同。如果 `count` 大于等于集合基数，那么返回整个集合。
- 如果 `count` 为负数，那么命令返回一个数组，数组中的元素可能会重复出现多次，而数组的长度为 `count` 的绝对值。

该操作和 *SPOP* 相似，但 *SPOP* 将随机元素从集合中移除并返回，而 SRANDMEMBER 则仅仅返回随机元素，而不对集合进行任何改动。

可用版本：

>= 1.0.0

时间复杂度:

只提供 `key` 参数时为 O(1) 。如果提供了 `count` 参数，那么为 O(N) ，N 为返回数组的元素个数。

返回值:

只提供 `key` 参数时，返回一个元素；如果集合为空，返回 `nil` 。如果提供了 `count` 参数，那么返回一个数组；如果集合为空，返回空数组。

```
# 添加元素

redis> SADD fruit apple banana cherry
(integer) 3

# 只给定 key 参数，返回一个随机元素

redis> SRANDMEMBER fruit
"cherry"

redis> SRANDMEMBER fruit
"apple"

# 给定 3 为 count 参数，返回 3 个随机元素
# 每个随机元素都不相同

redis> SRANDMEMBER fruit 3
1) "apple"
```

```
2) "banana"
3) "cherry"

# 给定 -3 为 count 参数，返回 3 个随机元素
# 元素可能会重复出现多次

redis> SRANDMEMBER fruit -3
1) "banana"
2) "cherry"
3) "apple"

redis> SRANDMEMBER fruit -3
1) "apple"
2) "apple"
3) "cherry"

# 如果 count 是整数，且大于等于集合基数，那么返回整个集合

redis> SRANDMEMBER fruit 10
1) "apple"
2) "banana"
3) "cherry"

# 如果 count 是负数，且 count 的绝对值大于集合的基数
# 那么返回的数组的长度为 count 的绝对值

redis> SRANDMEMBER fruit -10
1) "banana"
2) "apple"
3) "banana"
4) "cherry"
5) "apple"
6) "apple"
7) "cherry"
8) "apple"
9) "apple"
10) "banana"

# SRANDMEMBER 并不会修改集合内容

redis> SMEMBERS fruit
1) "apple"
2) "cherry"
3) "banana"

# 集合为空时返回 nil 或者空数组

redis> SRANDMEMBER not-exists
(nil)

redis> SRANDMEMBER not-eixsts 10
(empty list or set)
```

SREM

SREM key member [member ...]

移除集合 `key` 中的一个或多个 `member` 元素，不存在的 `member` 元素会被忽略。

当 `key` 不是集合类型，返回一个错误。

Note

在 Redis 2.4 版本以前， SREM 只接受单个 `member` 值。

可用版本：

>= 1.0.0

时间复杂度:

O(N)， `N` 为给定 `member` 元素的数量。

返回值:

被成功移除的元素的数量，不包括被忽略的元素。

```
# 测试数据

redis> SMEMBERS languages
1) "c"
2) "lisp"
3) "python"
4) "ruby"

# 移除单个元素

redis> SREM languages ruby
(integer) 1

# 移除不存在元素

redis> SREM languages non-exists-language
(integer) 0

# 移除多个元素

redis> SREM languages lisp python c
(integer) 3

redis> SMEMBERS languages
(empty list or set)
```

SUNION

SUNION key [key ...]

返回一个集合的全部成员，该集合是所有给定集合的并集。

不存在的 `key` 被视为空集。

可用版本：

>= 1.0.0

时间复杂度:

O(N)， `N` 是所有给定集合的成员数量之和。

返回值:

并集成员的列表。

```
redis> SMEMBERS songs
1) "Billie Jean"

redis> SMEMBERS my_songs
1) "Believe Me"

redis> SUNION songs my_songs
1) "Billie Jean"
2) "Believe Me"
```

SUNIONSTORE

SUNIONSTORE destination key [key ...]

这个命令类似于 *SUNION* 命令，但它将结果保存到 `destination` 集合，而不是简单地返回结果集。

如果 `destination` 已经存在，则将其覆盖。

`destination` 可以是 `key` 本身。

可用版本：

>= 1.0.0

时间复杂度:

O(N)， `N` 是所有给定集合的成员数量之和。

返回值:

结果集中的元素数量。

```
redis> SMEMBERS NoSQL
1) "MongoDB"
2) "Redis"

redis> SMEMBERS SQL
1) "sqlite"
2) "MySQL"

redis> SUNIONSTORE db NoSQL SQL
(integer) 4

redis> SMEMBERS db
1) "MySQL"
2) "sqlite"
3) "MongoDB"
4) "Redis"
```

SSCAN

SSCAN key cursor [MATCH pattern] [COUNT count]

详细信息请参考 *SCAN* 命令。

SortedSet（有序集合）

ZADD

ZADD key score member [[score member] [score member] ...]

将一个或多个 `member` 元素及其 `score` 值加入到有序集 `key` 当中。

如果某个 `member` 已经是有序集的成员，那么更新这个 `member` 的 `score` 值，并通过重新插入这个 `member` 元素，来保证该 `member` 在正确的位置上。

`score` 值可以是整数值或双精度浮点数。

如果 `key` 不存在，则创建一个空的有序集并执行 ZADD 操作。

当 `key` 存在但不是有序集类型时，返回一个错误。

对有序集的更多介绍请参见 sorted set 。

Note

在 Redis 2.4 版本以前， ZADD 每次只能添加一个元素。

可用版本：

>= 1.2.0

时间复杂度:

O(M*log(N))， `N` 是有序集的基数， `M` 为成功添加的新成员的数量。

返回值:

被成功添加的新成员的数量，不包括那些被更新的、已经存在的成员。

```
# 添加单个元素

redis> ZADD page_rank 10 google.com
(integer) 1

# 添加多个元素

redis> ZADD page_rank 9 baidu.com 8 bing.com
(integer) 2

redis> ZRANGE page_rank 0 -1 WITHSCORES
1) "bing.com"
2) "8"
3) "baidu.com"
4) "9"
5) "google.com"
6) "10"

# 添加已存在元素，且 score 值不变

redis> ZADD page_rank 10 google.com
(integer) 0

redis> ZRANGE page_rank 0 -1 WITHSCORES  # 没有改变
1) "bing.com"
2) "8"
3) "baidu.com"
4) "9"
5) "google.com"
6) "10"

# 添加已存在元素，但是改变 score 值

redis> ZADD page_rank 6 bing.com
(integer) 0

redis> ZRANGE page_rank 0 -1 WITHSCORES  # bing.com 元素的 score 值
1) "bing.com"
2) "6"
3) "baidu.com"
4) "9"
5) "google.com"
6) "10"
```

ZCARD

ZCARD key

返回有序集 `key` 的基数。

可用版本：

>= 1.2.0

时间复杂度:

O(1)

返回值:

当 `key` 存在且是有序集类型时，返回有序集的基数。当 `key` 不存在时，返回 `0` 。

```
redis > ZADD salary 2000 tom    # 添加一个成员
(integer) 1

redis > ZCARD salary
(integer) 1

redis > ZADD salary 5000 jack   # 再添加一个成员
(integer) 1

redis > ZCARD salary
(integer) 2

redis > EXISTS non_exists_key   # 对不存在的 key 进行 ZCARD 操作
(integer) 0

redis > ZCARD non_exists_key
(integer) 0
```

ZCOUNT

ZCOUNT key min max

返回有序集 `key` 中， `score` 值在 `min` 和 `max` 之间(默认包括 `score` 值等于 `min` 或 `max`)的成员的数量。

关于参数 `min` 和 `max` 的详细使用方法，请参考 *ZRANGEBYSCORE* 命令。

可用版本：

>= 2.0.0

时间复杂度:

O(log(N))， `N` 为有序集的基数。

返回值:

`score` 值在 `min` 和 `max` 之间的成员的数量。

```
redis> ZRANGE salary 0 -1 WITHSCORES    # 测试数据
1) "jack"
2) "2000"
3) "peter"
4) "3500"
5) "tom"
6) "5000"

redis> ZCOUNT salary 2000 5000          # 计算薪水在 2000-5000 之间的
(integer) 3

redis> ZCOUNT salary 3000 5000          # 计算薪水在 3000-5000 之间的
(integer) 2
```

ZINCRBY

ZINCRBY key increment member

为有序集 `key` 的成员 `member` 的 `score` 值加上增量 `increment` 。

可以通过传递一个负数值 `increment` ，让 `score` 减去相应的值，比如 `ZINCRBY key -5 member` ，就是让 `member` 的 `score` 值减去 `5` 。

当 `key` 不存在，或 `member` 不是 `key` 的成员时， `ZINCRBY key increment member` 等同于 `ZADD key increment member` 。

当 `key` 不是有序集类型时，返回一个错误。

`score` 值可以是整数值或双精度浮点数。

可用版本：

>= 1.2.0

时间复杂度:

O(log(N))

返回值:

`member` 成员的新 `score` 值，以字符串形式表示。

```
redis> ZSCORE salary tom
"2000"

redis> ZINCRBY salary 2000 tom   # tom 加薪啦！
"4000"
```

ZRANGE

ZRANGE key start stop [WITHSCORES]

返回有序集 `key` 中，指定区间内的成员。

其中成员的位置按 `score` 值递增(从小到大)来排序。

具有相同 `score` 值的成员按字典序(lexicographical order)来排列。

如果你需要成员按 `score` 值递减(从大到小)来排列，请使用 *ZREVRANGE* 命令。

下标参数 `start` 和 `stop` 都以 `0` 为底，也就是说，以 `0` 表示有序集第一个成员，以 `1` 表示有序集第二个成员，以此类推。你也可以使用负数下标，以 `-1` 表示最后一个成员， `-2` 表示倒数第二个成员，以此类推。超出范围的下标并不会引起错误。比如说，当 `start` 的值比有序集的最大下标还要大，或是 `start > stop` 时， ZRANGE 命令只是简单地返回一个空列表。另一方面，假如 `stop` 参数的值比有序集的最大下标还要大，那么 Redis 将 `stop` 当作最大下标来处理。可以通过使用 `WITHSCORES` 选项，来让成员和它的 `score` 值一并返回，返回列表以 `value1,score1, ..., valueN,scoreN` 的格式表示。客户端库可能会返回一些更复杂的数据类型，比如数组、元组等。

可用版本：

>= 1.2.0

时间复杂度:

O(log(N)+M)， `N` 为有序集的基数，而 `M` 为结果集的基数。

返回值:

指定区间内，带有 `score` 值(可选)的有序集成员的列表。

```
redis > ZRANGE salary 0 -1 WITHSCORES             # 显示整个有序集成
1) "jack"
2) "3500"
3) "tom"
4) "5000"
5) "boss"
6) "10086"

redis > ZRANGE salary 1 2 WITHSCORES              # 显示有序集下标区
1) "tom"
2) "5000"
3) "boss"
4) "10086"

redis > ZRANGE salary 0 200000 WITHSCORES         # 测试 end 下标超
1) "jack"
2) "3500"
3) "tom"
4) "5000"
5) "boss"
6) "10086"

redis > ZRANGE salary 200000 3000000 WITHSCORES   # 测试当给定区间不
(empty list or set)
```

ZRANGEBYSCORE

ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT offset count]

返回有序集 `key` 中，所有 `score` 值介于 `min` 和 `max` 之间(包括等于 `min` 或 `max`)的成员。有序集成员按 `score` 值递增(从小到大)次序排列。

具有相同 `score` 值的成员按字典序(lexicographical order)来排列(该属性是有序集提供的，不需要额外的计算)。

可选的 `LIMIT` 参数指定返回结果的数量及区间(就像SQL中的 `SELECT LIMIT offset, count`)，注意当 `offset` 很大时，定位 `offset` 的操作可能需要遍历整个有序集，此过程最坏复杂度为 O(N) 时间。

可选的 `WITHSCORES` 参数决定结果集是单单返回有序集的成员，还是将有序集成员及其 `score` 值一起返回。该选项自 Redis 2.0 版本起可用。

区间及无限

`min` 和 `max` 可以是 `-inf` 和 `+inf` ，这样一来，你就可以在不知道有序集的最低和最高 `score` 值的情况下，使用 ZRANGEBYSCORE 这类命令。

默认情况下，区间的取值使用闭区间 (小于等于或大于等于)，你也可以通过给参数前增加 `(` 符号来使用可选的开区间 (小于或大于)。

举个例子：

```
ZRANGEBYSCORE zset (1 5
```

返回所有符合条件 `1 < score <= 5` 的成员，而

```
ZRANGEBYSCORE zset (5 (10
```

则返回所有符合条件 `5 < score < 10` 的成员。

可用版本：

>= 1.0.5

时间复杂度:

O(log(N)+M)， `N` 为有序集的基数， `M` 为被结果集的基数。

返回值:

指定区间内，带有 `score` 值(可选)的有序集成员的列表。

```
redis> ZADD salary 2500 jack                        # 测试数据
(integer) 0
redis> ZADD salary 5000 tom
(integer) 0
redis> ZADD salary 12000 peter
(integer) 0

redis> ZRANGEBYSCORE salary -inf +inf               # 显示整个有序集
1) "jack"
2) "tom"
3) "peter"

redis> ZRANGEBYSCORE salary -inf +inf WITHSCORES    # 显示整个有序集及
1) "jack"
2) "2500"
3) "tom"
4) "5000"
5) "peter"
6) "12000"

redis> ZRANGEBYSCORE salary -inf 5000 WITHSCORES    # 显示工资 <=500
1) "jack"
2) "2500"
3) "tom"
4) "5000"

redis> ZRANGEBYSCORE salary (5000 400000            # 显示工资大于 50
1) "peter"
```

ZRANK

ZRANK key member

返回有序集 `key` 中成员 `member` 的排名。其中有序集成员按 `score` 值递增(从小到大)顺序排列。

排名以 `0` 为底，也就是说， `score` 值最小的成员排名为 `0` 。

使用 *ZREVRANK* 命令可以获得成员按 `score` 值递减(从大到小)排列的排名。

可用版本：

>= 2.0.0

时间复杂度:

O(log(N))

返回值:

如果 `member` 是有序集 `key` 的成员，返回 `member` 的排名。如果 `member` 不是有序集 `key` 的成员，返回 `nil` 。

```
redis> ZRANGE salary 0 -1 WITHSCORES        # 显示所有成员及其 score
1) "peter"
2) "3500"
3) "tom"
4) "4000"
5) "jack"
6) "5000"

redis> ZRANK salary tom                     # 显示 tom 的薪水排名，第
(integer) 1
```

ZREM

ZREM key member [member ...]

移除有序集 `key` 中的一个或多个成员，不存在的成员将被忽略。

当 `key` 存在但不是有序集类型时，返回一个错误。

Note

在 Redis 2.4 版本以前， ZREM 每次只能删除一个元素。

可用版本：

>= 1.2.0

时间复杂度:

O(M*log(N))， `N` 为有序集的基数， `M` 为被成功移除的成员的数量。

返回值:

被成功移除的成员的数量，不包括被忽略的成员。

```
# 测试数据

redis> ZRANGE page_rank 0 -1 WITHSCORES
1) "bing.com"
2) "8"
3) "baidu.com"
4) "9"
5) "google.com"
6) "10"

# 移除单个元素

redis> ZREM page_rank google.com
(integer) 1

redis> ZRANGE page_rank 0 -1 WITHSCORES
1) "bing.com"
2) "8"
3) "baidu.com"
4) "9"

# 移除多个元素

redis> ZREM page_rank baidu.com bing.com
(integer) 2

redis> ZRANGE page_rank 0 -1 WITHSCORES
(empty list or set)

# 移除不存在元素

redis> ZREM page_rank non-exists-element
(integer) 0
```

ZREMRANGEBYRANK

ZREMRANGEBYRANK key start stop

移除有序集 `key` 中，指定排名(rank)区间内的所有成员。

区间分别以下标参数 `start` 和 `stop` 指出，包含 `start` 和 `stop` 在内。

下标参数 `start` 和 `stop` 都以 `0` 为底，也就是说，以 `0` 表示有序集第一个成员，以 `1` 表示有序集第二个成员，以此类推。你也可以使用负数下标，以 `-1` 表示最后一个成员， `-2` 表示倒数第二个成员，以此类推。

可用版本：

>= 2.0.0

时间复杂度:

O(log(N)+M)， `N` 为有序集的基数，而 `M` 为被移除成员的数量。

返回值:

被移除成员的数量。

```
redis> ZADD salary 2000 jack
(integer) 1
redis> ZADD salary 5000 tom
(integer) 1
redis> ZADD salary 3500 peter
(integer) 1

redis> ZREMRANGEBYRANK salary 0 1       # 移除下标 0 至 1 区间内的成员
(integer) 2

redis> ZRANGE salary 0 -1 WITHSCORES    # 有序集只剩下一个成员
1) "tom"
2) "5000"
```

ZREMRANGEBYSCORE

ZREMRANGEBYSCORE key min max

移除有序集 `key` 中，所有 `score` 值介于 `min` 和 `max` 之间(包括等于 `min` 或 `max`)的成员。

自版本2.1.6开始， `score` 值等于 `min` 或 `max` 的成员也可以不包括在内，详情请参见 *ZRANGEBYSCORE* 命令。

可用版本：

>= 1.2.0

时间复杂度:

O(log(N)+M)， `N` 为有序集的基数，而 `M` 为被移除成员的数量。

返回值:

被移除成员的数量。

```
redis> ZRANGE salary 0 -1 WITHSCORES          # 显示有序集内所有成员及
1) "tom"
2) "2000"
3) "peter"
4) "3500"
5) "jack"
6) "5000"

redis> ZREMRANGEBYSCORE salary 1500 3500      # 移除所有薪水在 1500 到
(integer) 2

redis> ZRANGE salary 0 -1 WITHSCORES          # 剩下的有序集成员
1) "jack"
2) "5000"
```

ZREVRANGE

ZREVRANGE key start stop [WITHSCORES]

返回有序集 `key` 中，指定区间内的成员。

其中成员的位置按 `score` 值递减(从大到小)来排列。具有相同 `score` 值的成员按字典序的逆序(reverse lexicographical order)排列。

除了成员按 `score` 值递减的次序排列这一点外， ZREVRANGE 命令的其他方面和 *ZRANGE* 命令一样。

可用版本：

>= 1.2.0

时间复杂度:

O(log(N)+M)， `N` 为有序集的基数，而 `M` 为结果集的基数。

返回值:

指定区间内，带有 `score` 值(可选)的有序集成员的列表。

```
redis> ZRANGE salary 0 -1 WITHSCORES        # 递增排列
1) "peter"
2) "3500"
3) "tom"
4) "4000"
5) "jack"
6) "5000"

redis> ZREVRANGE salary 0 -1 WITHSCORES     # 递减排列
1) "jack"
2) "5000"
3) "tom"
4) "4000"
5) "peter"
6) "3500"
```

ZREVRANGEBYSCORE

ZREVRANGEBYSCORE key max min [WITHSCORES] [LIMIT offset count]

返回有序集 `key` 中， `score` 值介于 `max` 和 `min` 之间(默认包括等于 `max` 或 `min`)的所有的成员。有序集成员按 `score` 值递减(从大到小)的次序排列。

具有相同 `score` 值的成员按字典序的逆序(reverse lexicographical order)排列。

除了成员按 `score` 值递减的次序排列这一点外， ZREVRANGEBYSCORE 命令的其他方面和 *ZRANGEBYSCORE* 命令一样。

可用版本：

>= 2.2.0

时间复杂度:

O(log(N)+M)， `N` 为有序集的基数， `M` 为结果集的基数。

返回值:

指定区间内，带有 `score` 值(可选)的有序集成员的列表。

```
redis > ZADD salary 10086 jack
(integer) 1
redis > ZADD salary 5000 tom
(integer) 1
redis > ZADD salary 7500 peter
(integer) 1
redis > ZADD salary 3500 joe
(integer) 1

redis > ZREVRANGEBYSCORE salary +inf -inf   # 逆序排列所有成员
1) "jack"
2) "peter"
3) "tom"
4) "joe"

redis > ZREVRANGEBYSCORE salary 10000 2000  # 逆序排列薪水介于 10000
1) "peter"
2) "tom"
3) "joe"
```

ZREVRANK

ZREVRANK key member

返回有序集 `key` 中成员 `member` 的排名。其中有序集成员按 `score` 值递减(从大到小)排序。

排名以 `0` 为底，也就是说， `score` 值最大的成员排名为 `0` 。

使用 *ZRANK* 命令可以获得成员按 `score` 值递增(从小到大)排列的排名。

可用版本：

>= 2.0.0

时间复杂度:

O(log(N))

返回值:

如果 `member` 是有序集 `key` 的成员，返回 `member` 的排名。如果 `member` 不是有序集 `key` 的成员，返回 `nil` 。

```
redis 127.0.0.1:6379> ZRANGE salary 0 -1 WITHSCORES     # 测试数据
1) "jack"
2) "2000"
3) "peter"
4) "3500"
5) "tom"
6) "5000"

redis> ZREVRANK salary peter     # peter 的工资排第二
(integer) 1

redis> ZREVRANK salary tom       # tom 的工资最高
(integer) 0
```

ZSCORE

ZSCORE key member

返回有序集 `key` 中，成员 `member` 的 `score` 值。

如果 `member` 元素不是有序集 `key` 的成员，或 `key` 不存在，返回 `nil` 。

可用版本：

>= 1.2.0

时间复杂度:

O(1)

返回值:

`member` 成员的 `score` 值，以字符串形式表示。

```
redis> ZRANGE salary 0 -1 WITHSCORES    # 测试数据
1) "tom"
2) "2000"
3) "peter"
4) "3500"
5) "jack"
6) "5000"

redis> ZSCORE salary peter              # 注意返回值是字符串
"3500"
```

ZUNIONSTORE

ZUNIONSTORE destination numkeys key [key ...] [WEIGHTS weight [weight ...]] [AGGREGATE SUM|MIN|MAX]

计算给定的一个或多个有序集的并集，其中给定 `key` 的数量必须以 `numkeys` 参数指定，并将该并集(结果集)储存到 `destination` 。

默认情况下，结果集中某个成员的 `score` 值是所有给定集下该成员 `score` 值之 和 。

WEIGHTS

使用 `WEIGHTS` 选项，你可以为 每个 给定有序集 分别 指定一个乘法因子(multiplication factor)，每个给定有序集的所有成员的 `score` 值在传递给聚合函数(aggregation function)之前都要先乘以该有序集的因子。

如果没有指定 `WEIGHTS` 选项，乘法因子默认设置为 `1` 。

AGGREGATE

使用 `AGGREGATE` 选项，你可以指定并集的结果集的聚合方式。

默认使用的参数 `SUM` ，可以将所有集合中某个成员的 `score` 值之 和 作为结果集中该成员的 `score` 值；使用参数 `MIN` ，可以将所有集合中某个成员的 最小 `score` 值作为结果集中该成员的 `score` 值；而参数 `MAX` 则是将所有集合中某个成员的 最大 `score` 值作为结果集中该成员的 `score` 值。

可用版本：

>= 2.0.0

时间复杂度:

O(N)+O(M log(M))， `N` 为给定有序集基数的总和， `M` 为结果集的基数。

返回值:

保存到 `destination` 的结果集的基数。

```
redis> ZRANGE programmer 0 -1 WITHSCORES
1) "peter"
2) "2000"
3) "jack"
4) "3500"
5) "tom"
6) "5000"

redis> ZRANGE manager 0 -1 WITHSCORES
1) "herry"
2) "2000"
3) "mary"
4) "3500"
5) "bob"
6) "4000"

redis> ZUNIONSTORE salary 2 programmer manager WEIGHTS 1 3   # 公司
(integer) 6

redis> ZRANGE salary 0 -1 WITHSCORES
1) "peter"
2) "2000"
3) "jack"
4) "3500"
5) "tom"
6) "5000"
7) "herry"
8) "6000"
9) "mary"
10) "10500"
11) "bob"
12) "12000"
```

ZINTERSTORE

ZINTERSTORE destination numkeys key [key ...] [WEIGHTS weight [weight ...]] [AGGREGATE SUM|MIN|MAX]

计算给定的一个或多个有序集的交集，其中给定 `key` 的数量必须以 `numkeys` 参数指定，并将该交集(结果集)储存到 `destination` 。

默认情况下，结果集中某个成员的 `score` 值是所有给定集下该成员 `score` 值之和.

关于 `WEIGHTS` 和 `AGGREGATE` 选项的描述，参见 *ZUNIONSTORE* 命令。

可用版本：

>= 2.0.0

时间复杂度:

O(N*K)+O(M*log(M))， `N` 为给定 `key` 中基数最小的有序集， `K` 为给定有序集的数量， `M` 为结果集的基数。

返回值:

保存到 `destination` 的结果集的基数。

```
redis > ZADD mid_test 70 "Li Lei"
(integer) 1
redis > ZADD mid_test 70 "Han Meimei"
(integer) 1
redis > ZADD mid_test 99.5 "Tom"
(integer) 1

redis > ZADD fin_test 88 "Li Lei"
(integer) 1
redis > ZADD fin_test 75 "Han Meimei"
(integer) 1
redis > ZADD fin_test 99.5 "Tom"
(integer) 1

redis > ZINTERSTORE sum_point 2 mid_test fin_test
(integer) 3

redis > ZRANGE sum_point 0 -1 WITHSCORES     # 显示有序集内所有成员及其
1) "Han Meimei"
2) "145"
3) "Li Lei"
4) "158"
5) "Tom"
6) "199"
```

ZSCAN

ZSCAN key cursor [MATCH pattern] [COUNT count]

详细信息请参考 *SCAN* 命令。

Pub/Sub（发布/订阅）

PSUBSCRIBE

PSUBSCRIBE pattern [pattern ...]

订阅一个或多个符合给定模式的频道。

每个模式以 `*` 作为匹配符，比如 `it*` 匹配所有以 `it` 开头的频道(`it.news` 、 `it.blog` 、 `it.tweets` 等等)， `news.*` 匹配所有以 `news.` 开头的频道(`news.it` 、 `news.global.today` 等等)，诸如此类。

可用版本：

>= 2.0.0

时间复杂度：

O(N)， `N` 是订阅的模式的数量。

返回值：

接收到的信息(请参见下面的代码说明)。

```
# 订阅 news.* 和 tweet.* 两个模式

# 第 1 - 6 行是执行 psubscribe 之后的反馈信息
# 第 7 - 10 才是接收到的第一条信息
# 第 11 - 14 是第二条
# 以此类推。。。

redis> psubscribe news.* tweet.*
Reading messages... (press Ctrl-C to quit)
1) "psubscribe"                  # 返回值的类型：显示订阅成功
2) "news.*"                      # 订阅的模式
3) (integer) 1                   # 目前已订阅的模式的数量

1) "psubscribe"
2) "tweet.*"
3) (integer) 2

1) "pmessage"                    # 返回值的类型：信息
2) "news.*"                      # 信息匹配的模式
3) "news.it"                     # 信息本身的目标频道
4) "Google buy Motorola"         # 信息的内容

1) "pmessage"
2) "tweet.*"
3) "tweet.huangz"
4) "hello"

1) "pmessage"
2) "tweet.*"
3) "tweet.joe"
4) "@huangz morning"

1) "pmessage"
2) "news.*"
3) "news.life"
4) "An apple a day, keep doctors away"
```

PUBLISH

PUBLISH channel message

将信息 `message` 发送到指定的频道 `channel` 。

可用版本：

>= 2.0.0

时间复杂度：

O(N+M)，其中 `N` 是频道 `channel` 的订阅者数量，而 `M` 则是使用模式订阅(subscribed patterns)的客户端的数量。

返回值：

接收到信息 `message` 的订阅者数量。

```
# 对没有订阅者的频道发送信息

redis> publish bad_channel "can any body hear me?"
(integer) 0

# 向有一个订阅者的频道发送信息

redis> publish msg "good morning"
(integer) 1

# 向有多个订阅者的频道发送信息

redis> publish chat_room "hello~ everyone"
(integer) 3
```

PUBSUB

PUBSUB <subcommand> [argument [argument ...]]

PUBSUB 是一个查看订阅与发布系统状态的内省命令， 它由数个不同格式的子命令组成， 以下将分别对这些子命令进行介绍。

可用版本： >= 2.8.0

PUBSUB CHANNELS [pattern]

列出当前的活跃频道。

活跃频道指的是那些至少有一个订阅者的频道， 订阅模式的客户端不计算在内。

`pattern` 参数是可选的：

- 如果不给出 `pattern` 参数，那么列出订阅与发布系统中的所有活跃频道。
- 如果给出 `pattern` 参数，那么只列出和给定模式 `pattern` 相匹配的那些活跃频道。

复杂度： O(N) ， `N` 为活跃频道的数量（对于长度较短的频道和模式来说，将进行模式匹配的复杂度视为常数）。

返回值： 一个由活跃频道组成的列表。

```
# client-1 订阅 news.it 和 news.sport 两个频道

client-1> SUBSCRIBE news.it news.sport
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "news.it"
3) (integer) 1
1) "subscribe"
2) "news.sport"
3) (integer) 2

# client-2 订阅 news.it 和 news.internet 两个频道

client-2> SUBSCRIBE news.it news.internet
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "news.it"
3) (integer) 1
1) "subscribe"
2) "news.internet"
3) (integer) 2

# 首先， client-3 打印所有活跃频道
# 注意，即使一个频道有多个订阅者，它也只输出一次，比如 news.it

client-3> PUBSUB CHANNELS
1) "news.sport"
2) "news.internet"
3) "news.it"

# 接下来， client-3 打印那些与模式 news.i* 相匹配的活跃频道
# 因为 news.sport 不匹配 news.i* ，所以它没有被打印

redis> PUBSUB CHANNELS news.i*
1) "news.internet"
2) "news.it"
```

PUBSUB NUMSUB [channel-1 ... channel-N]

返回给定频道的订阅者数量， 订阅模式的客户端不计算在内。

复杂度： O(N) ， `N` 为给定频道的数量。

返回值： 一个多条批量回复（Multi-bulk reply），回复中包含给定的频道，以及频道的订阅者数量。格式为：频道 `channel-1` ， `channel-1` 的订阅者数量，频道 `channel-2` ， `channel-2` 的订阅者数量，诸如此类。回复中频道的排列顺序和执行命令时给定频道的排列顺序一致。不给定任何频道而直接调用这个命令也是可以的， 在这种情况下， 命令只返回一个空列表。

```
# client-1 订阅 news.it 和 news.sport 两个频道

client-1> SUBSCRIBE news.it news.sport
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "news.it"
3) (integer) 1
1) "subscribe"
2) "news.sport"
3) (integer) 2

# client-2 订阅 news.it 和 news.internet 两个频道

client-2> SUBSCRIBE news.it news.internet
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "news.it"
3) (integer) 1
1) "subscribe"
2) "news.internet"
3) (integer) 2

# client-3 打印各个频道的订阅者数量

client-3> PUBSUB NUMSUB news.it news.internet news.sport news.musi
1) "news.it"    # 频道
2) "2"          # 订阅该频道的客户端数量
3) "news.internet"
4) "1"
5) "news.sport"
6) "1"
7) "news.music" # 没有任何订阅者
8) "0"
```

PUBSUB NUMPAT

返回订阅模式的数量。

注意， 这个命令返回的不是订阅模式的客户端的数量， 而是客户端订阅的所有模式的数量总和。

复杂度： O(1) 。

返回值： 一个整数回复（Integer reply）。

```
# client-1 订阅 news.* 和 discount.* 两个模式

client-1> PSUBSCRIBE news.* discount.*
Reading messages... (press Ctrl-C to quit)
1) "psubscribe"
2) "news.*"
3) (integer) 1
1) "psubscribe"
2) "discount.*"
3) (integer) 2

# client-2 订阅 tweet.* 一个模式

client-2> PSUBSCRIBE tweet.*
Reading messages... (press Ctrl-C to quit)
1) "psubscribe"
2) "tweet.*"
3) (integer) 1

# client-3 返回当前订阅模式的数量为 3

client-3> PUBSUB NUMPAT
(integer) 3

# 注意，当有多个客户端订阅相同的模式时，相同的订阅也被计算在 PUBSUB NUMPAT 之
# 比如说，再新建一个客户端 client-4 ，让它也订阅 news.* 频道

client-4> PSUBSCRIBE news.*
Reading messages... (press Ctrl-C to quit)
1) "psubscribe"
2) "news.*"
3) (integer) 1

# 这时再计算被订阅模式的数量，就会得到数量为 4

client-3> PUBSUB NUMPAT
(integer) 4
```

PUNSUBSCRIBE

PUNSUBSCRIBE [pattern [pattern ...]]

指示客户端退订所有给定模式。

如果没有模式被指定，也即是，一个无参数的 `PUNSUBSCRIBE` 调用被执行，那么客户端使用 *PSUBSCRIBE* 命令订阅的所有模式都会被退订。在这种情况下，命令会返回一个信息，告知客户端所有被退订的模式。

可用版本：

>= 2.0.0

时间复杂度：

O(N+M)，其中 `N` 是客户端已订阅的模式的数量， `M` 则是系统中所有客户端订阅的模式的数量。

返回值：

这个命令在不同的客户端中有不同的表现。

SUBSCRIBE

SUBSCRIBE channel [channel ...]

订阅给定的一个或多个频道的信息。

可用版本：

>= 2.0.0

时间复杂度：

O(N)，其中 `N` 是订阅的频道的数量。

返回值：

接收到的信息(请参见下面的代码说明)。

```
# 订阅 msg 和 chat_room 两个频道

# 1 - 6 行是执行 subscribe 之后的反馈信息
# 第 7 - 9 行才是接收到的第一条信息
# 第 10 - 12 行是第二条

redis> subscribe msg chat_room
Reading messages... (press Ctrl-C to quit)
1) "subscribe"       # 返回值的类型：显示订阅成功
2) "msg"             # 订阅的频道名字
3) (integer) 1       # 目前已订阅的频道数量

1) "subscribe"
2) "chat_room"
3) (integer) 2

1) "message"         # 返回值的类型：信息
2) "msg"             # 来源(从那个频道发送过来)
3) "hello moto"      # 信息内容

1) "message"
2) "chat_room"
3) "testing...haha"
```

UNSUBSCRIBE

UNSUBSCRIBE [channel [channel ...]]

指示客户端退订给定的频道。

如果没有频道被指定，也即是，一个无参数的 `UNSUBSCRIBE` 调用被执行，那么客户端使用 *SUBSCRIBE* 命令订阅的所有频道都会被退订。在这种情况下，命令会返回一个信息，告知客户端所有被退订的频道。

可用版本：

>= 2.0.0

时间复杂度：

O(N)， `N` 是客户端已订阅的频道的数量。

返回值：

这个命令在不同的客户端中有不同的表现。

Transaction（事务）

DISCARD

DISCARD

取消事务，放弃执行事务块内的所有命令。

如果正在使用 *WATCH* 命令监视某个(或某些) key，那么取消所有监视，等同于执行命令 *UNWATCH* 。

可用版本：

>= 2.0.0

时间复杂度：

O(1)。

返回值：

总是返回 `OK` 。

```
redis> MULTI
OK

redis> PING
QUEUED

redis> SET greeting "hello"
QUEUED

redis> DISCARD
OK
```

EXEC

EXEC

执行所有事务块内的命令。

假如某个(或某些) key 正处于 WATCH 命令的监视之下，且事务块中有和这个(或这些) key 相关的命令，那么 EXEC 命令只在这个(或这些) key 没有被其他命令所改动的情况下执行并生效，否则该事务被打断(abort)。

可用版本：

>= 1.2.0

时间复杂度：

事务块内所有命令的时间复杂度的总和。

返回值：

事务块内所有命令的返回值，按命令执行的先后顺序排列。当操作被打断时，返回空值 `nil` 。

```
# 事务被成功执行

redis> MULTI
OK

redis> INCR user_id
QUEUED

redis> INCR user_id
QUEUED

redis> INCR user_id
QUEUED

redis> PING
QUEUED

redis> EXEC
1) (integer) 1
2) (integer) 2
3) (integer) 3
4) PONG

# 监视 key ，且事务成功执行

redis> WATCH lock lock_times
OK
```

```
redis> MULTI
OK

redis> SET lock "huangz"
QUEUED

redis> INCR lock_times
QUEUED

redis> EXEC
1) OK
2) (integer) 1

# 监视 key ，且事务被打断

redis> WATCH lock lock_times
OK

redis> MULTI
OK

redis> SET lock "joe"        # 就在这时，另一个客户端修改了 lock_times
QUEUED

redis> INCR lock_times
QUEUED

redis> EXEC                  # 因为 lock_times 被修改， joe 的事务执行
(nil)
```

MULTI

MULTI

标记一个事务块的开始。

事务块内的多条命令会按照先后顺序被放进一个队列当中，最后由 *EXEC* 命令原子性(atomic)地执行。

可用版本：

>= 1.2.0

时间复杂度：

O(1)。

返回值：

总是返回 `OK` 。

```
redis> MULTI                # 标记事务开始
OK

redis> INCR user_id         # 多条命令按顺序入队
QUEUED

redis> INCR user_id
QUEUED

redis> INCR user_id
QUEUED

redis> PING
QUEUED

redis> EXEC                 # 执行
1) (integer) 1
2) (integer) 2
3) (integer) 3
4) PONG
```

UNWATCH

UNWATCH

取消 *WATCH* 命令对所有 key 的监视。

如果在执行 *WATCH* 命令之后， *EXEC* 命令或 *DISCARD* 命令先被执行了的话，那么就不需要再执行 UNWATCH 了。

因为 *EXEC* 命令会执行事务，因此 *WATCH* 命令的效果已经产生了；而 *DISCARD* 命令在取消事务的同时也会取消所有对 key 的监视，因此这两个命令执行之后，就没有必要执行 UNWATCH 了。

可用版本：

>= 2.2.0

时间复杂度：

O(1)

返回值：

总是 `OK` 。

```
redis> WATCH lock lock_times
OK

redis> UNWATCH
OK
```

WATCH

WATCH key [key ...]

监视一个(或多个) key ，如果在事务执行之前这个(或这些) key 被其他命令所改动，那么事务将被打断。

可用版本：

>= 2.2.0

时间复杂度：

O(1)。

返回值：

总是返回 `OK` 。

```
redis> WATCH lock lock_times
OK
```

Script（脚本）

EVAL

EVAL script numkeys key [key ...] arg [arg ...]

从 Redis 2.6.0 版本开始，通过内置的 Lua 解释器，可以使用 EVAL 命令对 Lua 脚本进行求值。

`script` 参数是一段 Lua 5.1 脚本程序，它会被运行在 Redis 服务器上下文中，这段脚本不必(也不应该)定义为一个 Lua 函数。

`numkeys` 参数用于指定键名参数的个数。

键名参数 `key [key ...]` 从 EVAL 的第三个参数开始算起，表示在脚本中所用到的那些 Redis 键(key)，这些键名参数可以在 Lua 中通过全局变量 `KEYS` 数组，用 `1` 为基址的形式访问(`KEYS[1]` ， `KEYS[2]` ，以此类推)。

在命令的最后，那些不是键名参数的附加参数 `arg [arg ...]` ，可以在 Lua 中通过全局变量 `ARGV` 数组访问，访问的形式和 `KEYS` 变量类似(`ARGV[1]` 、 `ARGV[2]` ，诸如此类)。

上面这几段长长的说明可以用一个简单的例子来概括：

```
> eval "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}" 2 key1 key2 first
1) "key1"
2) "key2"
3) "first"
4) "second"
```

其中 `"return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}"` 是被求值的 Lua 脚本，数字 `2` 指定了键名参数的数量， `key1` 和 `key2` 是键名参数，分别使用 `KEYS[1]` 和 `KEYS[2]` 访问，而最后的 `first` 和 `second` 则是附加参数，可以通过 `ARGV[1]` 和 `ARGV[2]` 访问它们。

在 Lua 脚本中，可以使用两个不同函数来执行 Redis 命令，它们分别是：

- `redis.call()`
- `redis.pcall()`

这两个函数的唯一区别在于它们使用不同的方式处理执行命令所产生的错误，在后面的『错误处理』部分会讲到这一点。

`redis.call()` 和 `redis.pcall()` 两个函数的参数可以是任何格式良好(well formed)的 Redis 命令：

```
> eval "return redis.call('set','foo','bar')" 0
OK
```

需要注意的是，上面这段脚本的确实现了将键 `foo` 的值设为 `bar` 的目的，但是，它违反了 EVAL 命令的语义，因为脚本里使用的所有键都应该由 `KEYS` 数组来传递，就像这样：

```
> eval "return redis.call('set',KEYS[1],'bar')" 1 foo
OK
```

要求使用正确的形式来传递键(key)是有原因的，因为不仅仅是 EVAL 这个命令，所有的 Redis 命令，在执行之前都会被分析，籍此来确定命令会对哪些键进行操作。

因此，对于 EVAL 命令来说，必须使用正确的形式来传递键，才能确保分析工作正确地执行。除此之外，使用正确的形式来传递键还有很多其他好处，它的一个特别重要的用途就是确保 Redis 集群可以将你的请求发送到正确的集群节点。(对 Redis 集群的工作还在进行当中，但是脚本功能被设计成可以与集群功能保持兼容。)不过，这条规矩并不是强制性的，从而使得用户有机会滥用(abuse) Redis 单实例配置(single instance configuration)，代价是这样写出的脚本不能被 Redis 集群所兼容。

在 Lua 数据类型和 Redis 数据类型之间转换

当 Lua 通过 `call()` 或 `pcall()` 函数执行 Redis 命令的时候，命令的返回值会被转换成 Lua 数据结构。同样地，当 Lua 脚本在 Redis 内置的解释器里运行时，Lua 脚本的返回值也会被转换成 Redis 协议(protocol)，然后由 EVAL 将值返回给客户端。

数据类型之间的转换遵循这样一个设计原则：如果将一个 Redis 值转换成 Lua 值，之后再将转换所得的 Lua 值转换回 Redis 值，那么这个转换所得的 Redis 值应该和最初时的 Redis 值一样。

换句话说， Lua 类型和 Redis 类型之间存在着一一对应的转换关系。

以下列出的是详细的转换规则：

从 Redis 转换到 Lua ：

- Redis integer reply -> Lua number / Redis 整数转换成 Lua 数字
- Redis bulk reply -> Lua string / Redis bulk 回复转换成 Lua 字符串
- Redis multi bulk reply -> Lua table (may have other Redis data types nested) / Redis 多条 bulk 回复转换成 Lua 表，表内可能有其他别的 Redis 数据类型
- Redis status reply -> Lua table with a single ok field containing the status / Redis 状态回复转换成 Lua 表，表内的 `ok` 域包含了状态信息
- Redis error reply -> Lua table with a single err field containing the error / Redis 错误回复转换成 Lua 表，表内的 `err` 域包含了错误信息

- Redis Nil bulk reply and Nil multi bulk reply -> Lua false boolean type / Redis 的 Nil 回复和 Nil 多条回复转换成 Lua 的布尔值 `false`

从 Lua 转换到 Redis：

- Lua number -> Redis integer reply / Lua 数字转换成 Redis 整数
- Lua string -> Redis bulk reply / Lua 字符串转换成 Redis bulk 回复
- Lua table (array) -> Redis multi bulk reply / Lua 表(数组)转换成 Redis 多条 bulk 回复
- Lua table with a single ok field -> Redis status reply / 一个带单个 `ok` 域的 Lua 表，转换成 Redis 状态回复
- Lua table with a single err field -> Redis error reply / 一个带单个 `err` 域的 Lua 表，转换成 Redis 错误回复
- Lua boolean false -> Redis Nil bulk reply / Lua 的布尔值 `false` 转换成 Redis 的 Nil bulk 回复

从 Lua 转换到 Redis 有一条额外的规则，这条规则没有和它对应的从 Redis 转换到 Lua 的规则：

- Lua boolean true -> Redis integer reply with value of 1 / Lua 布尔值 `true` 转换成 Redis 整数回复中的 `1`

以下是几个类型转换的例子：

```
> eval "return 10" 0
(integer) 10

> eval "return {1,2,{3,'Hello World!'}}" 0
1) (integer) 1
2) (integer) 2
3) 1) (integer) 3
   2) "Hello World!"

> eval "return redis.call('get','foo')" 0
"bar"
```

在上面的三个代码示例里，前两个演示了如何将 Lua 值转换成 Redis 值，最后一个例子更复杂一些，它演示了一个将 Redis 值转换成 Lua 值，然后再将 Lua 值转换成 Redis 值的类型转过程。

脚本的原子性

Redis 使用单个 Lua 解释器去运行所有脚本，并且， Redis 也保证脚本会以原子性(atomic)的方式执行：当某个脚本正在运行的时候，不会有其他脚本或 Redis 命令被执行。这和使用 *MULTI* / *EXEC* 包围的事务很类似。在其他别的客户端看来，脚本的效果(effect)要么是不可见的(not visible)，要么就是已完成的(already completed)。

另一方面，这也意味着，执行一个运行缓慢的脚本并不是一个好主意。写一个跑得很快很顺溜的脚本并不难，因为脚本的运行开销(overhead)非常少，但是当你不得不使用一些跑得比较慢的脚本时，请小心，因为当这些蜗牛脚本在慢吞吞地运行的时候，其他客户端会因为服务器正忙而无法执行命令。

错误处理

前面的命令介绍部分说过， `redis.call()` 和 `redis.pcall()` 的唯一区别在于它们对错误处理的不同。

当 `redis.call()` 在执行命令的过程中发生错误时，脚本会停止执行，并返回一个脚本错误，错误的输出信息会说明错误造成的原因：

```
redis> lpush foo a
(integer) 1

redis> eval "return redis.call('get', 'foo')" 0
(error) ERR Error running script (call to f_282297a0228f48cd3fc6a55
```

和 `redis.call()` 不同， `redis.pcall()` 出错时并不引发(raise)错误，而是返回一个带 `err` 域的 Lua 表(table)，用于表示错误：

```
redis 127.0.0.1:6379> EVAL "return redis.pcall('get', 'foo')" 0
(error) ERR Operation against a key holding the wrong kind of value
```

带宽和 EVALSHA

EVAL 命令要求你在每次执行脚本的时候都发送一次脚本主体(script body)。Redis 有一个内部的缓存机制，因此它不会每次都重新编译脚本，不过在很多场合，付出无谓的带宽来传送脚本主体并不是最佳选择。

为了减少带宽的消耗， Redis 实现了 EVALSHA 命令，它的作用和 EVAL 一样，都用于对脚本求值，但它接受的第一个参数不是脚本，而是脚本的 SHA1 校验和(sum)。

EVALSHA 命令的表现如下：

- 如果服务器还记得给定的 SHA1 校验和所指定的脚本，那么执行这个脚本
- 如果服务器不记得给定的 SHA1 校验和所指定的脚本，那么它返回一个特殊的错误，提醒用户使用 EVAL 代替 EVALSHA

以下是示例：

```
> set foo bar
OK

> eval "return redis.call('get','foo')" 0
"bar"

> evalsha 6b1bf486c81ceb7edf3c093f4c48582e38c0e791 0
"bar"

> evalsha ffffffffffffffffffffffffffffffffffffffff 0
(error) `NOSCRIPT` No matching script. Please use [EVAL](/commands/
```

客户端库的底层实现可以一直乐观地使用 EVALSHA 来代替 EVAL，并期望着要使用的脚本已经保存在服务器上了，只有当 `NOSCRIPT` 错误发生时，才使用 EVAL 命令重新发送脚本，这样就可以最大限度地节省带宽。

这也说明了执行 EVAL 命令时，使用正确的格式来传递键名参数和附加参数的重要性：因为如果将参数硬写在脚本中，那么每次当参数改变的时候，都要重新发送脚本，即使脚本的主体并没有改变，相反，通过使用正确的格式来传递键名参数和附加参数，就可以在脚本主体不变的情况下，直接使用 EVALSHA 命令对脚本进行复用，免去了无谓的带宽消耗。

脚本缓存

Redis 保证所有被运行过的脚本都会被永久保存在脚本缓存当中，这意味着，当 EVAL 命令在一个 Redis 实例上成功执行某个脚本之后，随后针对这个脚本的所有 EVALSHA 命令都会成功执行。

刷新脚本缓存的唯一办法是显式地调用 `SCRIPT FLUSH` 命令，这个命令会清空运行过的所有脚本的缓存。通常只有在云计算环境中，Redis 实例被改作其他客户或者别的应用程序的实例时，才会执行这个命令。

缓存可以长时间储存而不产生内存问题的原因是，它们的体积非常小，而且数量也非常少，即使脚本在概念上类似于实现一个新命令，即使在一个大规模的程序里有成百上千的脚本，即使这些脚本会经常修改，即便如此，储存这些脚本的内存仍然是微不足道的。

事实上，用户会发现 Redis 不移除缓存中的脚本实际上是一个好主意。比如说，对于一个和 Redis 保持持久化链接(persistent connection)的程序来说，它可以确信，执行过一次的脚本会一直保留在内存当中，因此它可以在流水线中使用 EVALSHA 命令而不必担心因为找不到所需的脚本而产生错误(稍候我们会看到在流水线中执行脚本的相关问题)。

SCRIPT 命令

Redis 提供了以下几个 SCRIPT 命令，用于对脚本子系统(scripting subsystem)进行控制：

- *SCRIPT FLUSH* ：清除所有脚本缓存
- *SCRIPT EXISTS* ：根据给定的脚本校验和，检查指定的脚本是否存在于脚本缓存
- *SCRIPT LOAD* ：将一个脚本装入脚本缓存，但并不立即运行它
- *SCRIPT KILL* ：杀死当前正在运行的脚本

纯函数脚本

在编写脚本方面，一个重要的要求就是，脚本应该被写成纯函数(pure function)。

也就是说，脚本应该具有以下属性：

- 对于同样的数据集输入，给定相同的参数，脚本执行的 Redis 写命令总是相同的。脚本执行的操作不能依赖于任何隐藏(非显式)数据，不能依赖于脚本在执行过程中、或脚本在不同执行时期之间可能变更的状态，并且它也不能依赖于任何来自 I/O 设备的外部输入。

使用系统时间(system time)，调用像 *RANDOMKEY* 那样的随机命令，或者使用 Lua 的随机数生成器，类似以上的这些操作，都会造成脚本的求值无法每次都得出同样的结果。

为了确保脚本符合上面所说的属性， Redis 做了以下工作：

- Lua 没有访问系统时间或者其他内部状态的命令
- Redis 会返回一个错误，阻止这样的脚本运行： 这些脚本在执行随机命令之后(比如 *RANDOMKEY* 、 *SRANDMEMBER* 或 *TIME* 等)，还会执行可以修改数据集的 Redis 命令。如果脚本只是执行只读操作，那么就没有这一限制。注意，随机命令并不一定就指那些带 RAND 字眼的命令，任何带有非确定性的命令都会被认为是随机命令，比如 *TIME* 命令就是这方面的一个很好的例子。
- 每当从 Lua 脚本中调用那些返回无序元素的命令时，执行命令所得的数据在返回给 Lua 之前会先执行一个静默(slient)的字典序排序(lexicographical sorting)。举个例子，因为 Redis 的 Set 保存的是无序的元素，所以在 Redis 命令行客户端中直接执行 *SMEMBERS* ，返回的元素是无序的，但是，假如在脚本中执行 `redis.call("smembers", KEYS[1])` ，那么返回的总是排过序的元素。
- 对 Lua 的伪随机数生成函数 `math.random` 和 `math.randomseed` 进行修改，使得每次在运行新脚本的时候，总是拥有同样的 seed 值。这意味着，每次运行脚本时，只要不使用 `math.randomseed` ，那么 `math.random` 产生的随机数序列总是相同的。

尽管有那么多的限制，但用户还是可以用一个简单的技巧写出带随机行为的脚本(如果他们需要的话)。

假设现在我们要编写一个 Redis 脚本，这个脚本从列表中弹出 N 个随机数。一个 Ruby 写的例子如下：

```
require 'rubygems'
require 'redis'

r = Redis.new

RandomPushScript = <<EOF
    local i = tonumber(ARGV[1])
    local res
    while (i > 0) do
        res = redis.call('lpush',KEYS[1],math.random())
        i = i-1
    end
    return res
EOF

r.del(:mylist)
puts r.eval(RandomPushScript,[:mylist],[10,rand(2**32)])
```

这个程序每次运行都会生成带有以下元素的列表：

```
> lrange mylist 0 -1
1) "0.74509509873814"
2) "0.87390407681181"
3) "0.36876626981831"
4) "0.6921941534114"
5) "0.7857992587545"
6) "0.57730350670279"
7) "0.87046522734243"
8) "0.09637165539729"
9) "0.74990198051087"
10) "0.17082803611217"
```

上面的 Ruby 程序每次都只生成同样的列表，用途并不是太大。那么，该怎样修改这个脚本，使得它仍然是一个纯函数(符合 Redis 的要求)，但是每次调用都可以产生不同的随机元素呢？

一个简单的办法是，为脚本添加一个额外的参数，让这个参数作为 Lua 的随机数生成器的 seed 值，这样的话，只要给脚本传入不同的 seed，脚本就会生成不同的列表元素。

以下是修改后的脚本：

```
RandomPushScript = <<EOF
    local i = tonumber(ARGV[1])
    local res
    math.randomseed(tonumber(ARGV[2]))
    while (i > 0) do
        res = redis.call('lpush',KEYS[1],math.random())
        i = i-1
    end
    return res
EOF

r.del(:mylist)
puts r.eval(RandomPushScript,1,:mylist,10,rand(2**32))
```

尽管对于同样的 seed ，上面的脚本产生的列表元素是一样的(因为它是一个纯函数)，但是只要每次在执行脚本的时候传入不同的 seed ，我们就可以得到带有不同随机元素的列表。

Seed 会在复制(replication link)和写 AOF 文件时作为一个参数来传播，保证在载入 AOF 文件或附属节点(slave)处理脚本时， seed 仍然可以及时得到更新。

注意，Redis 实现保证 `math.random` 和 `math.randomseed` 的输出和运行 Redis 的系统架构无关，无论是 32 位还是 64 位系统，无论是小端(little endian)还是大端(big endian)系统，这两个函数的输出总是相同的。

全局变量保护

为了防止不必要的数据泄漏进 Lua 环境， Redis 脚本不允许创建全局变量。如果一个脚本需要在多次执行之间维持某种状态，它应该使用 Redis key 来进行状态保存。

企图在脚本中访问一个全局变量(不论这个变量是否存在)将引起脚本停止， EVAL 命令会返回一个错误：

```
redis 127.0.0.1:6379> eval 'a=10' 0
(error) ERR Error running script (call to f_933044db579a2f8fd45d806
```

Lua 的 debug 工具，或者其他设施，比如打印（alter）用于实现全局保护的 meta table ，都可以用于实现全局变量保护。

实现全局变量保护并不难，不过有时候还是会不小心而为之。一旦用户在脚本中混入了 Lua 全局状态，那么 AOF 持久化和复制（replication）都会无法保证，所以，请不要使用全局变量。

避免引入全局变量的一个诀窍是：将脚本中用到的所有变量都使用 `local` 关键字定义为局部变量。

库

Redis 内置的 Lua 解释器加载了以下 Lua 库：

- `base`
- `table`
- `string`
- `math`
- `debug`
- `cjson`
- `cmsgpack`

其中 `cjson` 库可以让 Lua 以非常快的速度处理 JSON 数据，除此之外，其他别的都是 Lua 的标准库。

每个 Redis 实例都保证会加载上面列举的库，从而确保每个 Redis 脚本的运行环境都是相同的。

使用脚本散发 **Redis** 日志

在 Lua 脚本中，可以通过调用 `redis.log` 函数来写 Redis 日志(log)：

```
redis.log(loglevel, message)
```

其中， `message` 参数是一个字符串，而 `loglevel` 参数可以是以下任意一个值：

- `redis.LOG_DEBUG`
- `redis.LOG_VERBOSE`
- `redis.LOG_NOTICE`
- `redis.LOG_WARNING`

上面的这些等级(level)和标准 Redis 日志的等级相对应。

对于脚本散发(emit)的日志，只有那些和当前 Redis 实例所设置的日志等级相同或更高级的日志才会被散发。

以下是一个日志示例：

```
redis.log(redis.LOG_WARNING, "Something is wrong with this script."
```

执行上面的函数会产生这样的信息：

```
[32343] 22 Mar 15:21:39 # Something is wrong with this script.
```

沙箱(**sandbox**)和最大执行时间

脚本应该仅仅用于传递参数和对 Redis 数据进行处理，它不应该尝试去访问外部系统(比如文件系统)，或者执行任何系统调用。

除此之外，脚本还有一个最大执行时间限制，它的默认值是 5 秒钟，一般正常运作的脚本通常可以在几分之几毫秒之内完成，花不了那么多时间，这个限制主要是为了防止因编程错误而造成的无限循环而设置的。

最大执行时间的长短由 `lua-time-limit` 选项来控制(以毫秒为单位)，可以通过编辑 `redis.conf` 文件或者使用 *CONFIG GET* 和 *CONFIG SET* 命令来修改它。

当一个脚本达到最大执行时间的时候，它并不会自动被 Redis 结束，因为 Redis 必须保证脚本执行的原子性，而中途停止脚本的运行意味着可能会留下未处理完的数据在数据集(data set)里面。

因此，当脚本运行的时间超过最大执行时间后，以下动作会被执行：

- Redis 记录一个脚本正在超时运行
- Redis 开始重新接受其他客户端的命令请求，但是只有 `SCRIPT KILL` 和 `SHUTDOWN NOSAVE` 两个命令会被处理，对于其他命令请求， Redis 服务器只是简单地返回 `BUSY` 错误。
- 可以使用 `SCRIPT KILL` 命令将一个仅执行只读命令的脚本杀死，因为只读命令并不修改数据，因此杀死这个脚本并不破坏数据的完整性
- 如果脚本已经执行过写命令，那么唯一允许执行的操作就是 `SHUTDOWN NOSAVE` ，它通过停止服务器来阻止当前数据集写入磁盘

流水线(pipeline)上下文(context)中的 EVALSHA

在流水线请求的上下文中使用 EVALSHA 命令时，要特别小心，因为在流水线中，必须保证命令的执行顺序。

一旦在流水线中因为 EVALSHA 命令而发生 NOSCRIPT 错误，那么这个流水线就再也没有办法重新执行了，否则的话，命令的执行顺序就会被打乱。

为了防止出现以上所说的问题，客户端库实现应该实施以下的其中一项措施：

- 总是在流水线中使用 EVAL 命令
- 检查流水线中要用到的所有命令，找到其中的 EVAL 命令，并使用 *SCRIPT EXISTS* 命令检查要用到的脚本是不是全都已经保存在缓存里面了。如果所需的全部脚本都可以在缓存里找到，那么就可以放心地将所有 EVAL 命令改成 EVALSHA 命令，否则的话，就要在流水线的顶端(top)将缺少的脚本用 *SCRIPT LOAD* 命令加上去。

可用版本：

>= 2.6.0

时间复杂度：

EVAL 和 EVALSHA 可以在 O(1) 复杂度内找到要被执行的脚本，其余的复杂度取决于执行的脚本本身。

EVALSHA

EVALSHA sha1 numkeys key [key ...] arg [arg ...]

根据给定的 sha1 校验码，对缓存在服务器中的脚本进行求值。

将脚本缓存到服务器的操作可以通过 *SCRIPT LOAD* 命令进行。

这个命令的其他地方，比如参数的传入方式，都和 *EVAL* 命令一样。

可用版本：

>= 2.6.0

时间复杂度：

根据脚本的复杂度而定。

```
redis> SCRIPT LOAD "return 'hello moto'"
"232fd51614574cf0867b83d384a5e898cfd24e5a"

redis> EVALSHA "232fd51614574cf0867b83d384a5e898cfd24e5a" 0
"hello moto"
```

SCRIPT EXISTS

SCRIPT EXISTS script [script ...]

给定一个或多个脚本的 SHA1 校验和，返回一个包含 `0` 和 `1` 的列表，表示校验和所指定的脚本是否已经被保存在缓存当中。

关于使用 Redis 对 Lua 脚本进行求值的更多信息，请参见 *EVAL* 命令。

可用版本：

>= 2.6.0

时间复杂度：

O(N) , `N` 为给定的 SHA1 校验和的数量。

返回值：

一个列表，包含 `0` 和 `1` ，前者表示脚本不存在于缓存，后者表示脚本已经在缓存里面了。列表中的元素和给定的 SHA1 校验和保持对应关系，比如列表的第三个元素的值就表示第三个 SHA1 校验和所指定的脚本在缓存中的状态。

```
redis> SCRIPT LOAD "return 'hello moto'"    # 载入一个脚本
"232fd51614574cf0867b83d384a5e898cfd24e5a"

redis> SCRIPT EXISTS 232fd51614574cf0867b83d384a5e898cfd24e5a
1) (integer) 1

redis> SCRIPT FLUSH     # 清空缓存
OK

redis> SCRIPT EXISTS 232fd51614574cf0867b83d384a5e898cfd24e5a
1) (integer) 0
```

SCRIPT FLUSH

SCRIPT FLUSH

清除所有 Lua 脚本缓存。

关于使用 Redis 对 Lua 脚本进行求值的更多信息，请参见 *EVAL* 命令。

可用版本：

>= 2.6.0

复杂度：

O(N)， `N` 为缓存中脚本的数量。

返回值：

总是返回 `OK`

```
redis> SCRIPT FLUSH
OK
```

SCRIPT KILL

SCRIPT KILL

杀死当前正在运行的 Lua 脚本，当且仅当这个脚本没有执行过任何写操作时，这个命令才生效。

这个命令主要用于终止运行时间过长的脚本，比如一个因为 BUG 而发生无限 loop 的脚本，诸如此类。

SCRIPT KILL 执行之后，当前正在运行的脚本会被杀死，执行这个脚本的客户端会从 *EVAL* 命令的阻塞当中退出，并收到一个错误作为返回值。

另一方面，假如当前正在运行的脚本已经执行过写操作，那么即使执行 SCRIPT KILL ，也无法将它杀死，因为这是违反 Lua 脚本的原子性执行原则的。在这种情况下，唯一可行的办法是使用 `SHUTDOWN NOSAVE` 命令，通过停止整个 Redis 进程来停止脚本的运行，并防止不完整(half-written)的信息被写入数据库中。

关于使用 Redis 对 Lua 脚本进行求值的更多信息，请参见 *EVAL* 命令。

可用版本：

>= 2.6.0

时间复杂度：

O(1)

返回值：

执行成功返回 `OK` ，否则返回一个错误。

```
# 没有脚本在执行时

redis> SCRIPT KILL
(error) ERR No scripts in execution right now.

# 成功杀死脚本时

redis> SCRIPT KILL
OK
(1.30s)

# 尝试杀死一个已经执行过写操作的脚本，失败

redis> SCRIPT KILL
(error) ERR Sorry the script already executed write commands agains
(1.69s)
```

以下是脚本被杀死之后，返回给执行脚本的客户端的错误：

```
redis> EVAL "while true do end" 0
(error) ERR Error running script (call to f_694a5fe1ddb97a4c6a1bf29
(5.00s)
```

SCRIPT LOAD

SCRIPT LOAD script

将脚本 `script` 添加到脚本缓存中，但并不立即执行这个脚本。

EVAL 命令也会将脚本添加到脚本缓存中，但是它会立即对输入的脚本进行求值。

如果给定的脚本已经在缓存里面了，那么不做动作。

在脚本被加入到缓存之后，通过 EVALSHA 命令，可以使用脚本的 SHA1 校验和来调用这个脚本。

脚本可以在缓存中保留无限长的时间，直到执行 *SCRIPT FLUSH* 为止。

关于使用 Redis 对 Lua 脚本进行求值的更多信息，请参见 *EVAL* 命令。

可用版本：

>= 2.6.0

时间复杂度：

O(N) , `N` 为脚本的长度(以字节为单位)。

返回值：

给定 `script` 的 SHA1 校验和

```
redis> SCRIPT LOAD "return 'hello moto'"
"232fd51614574cf0867b83d384a5e898cfd24e5a"

redis> EVALSHA 232fd51614574cf0867b83d384a5e898cfd24e5a 0
"hello moto"
```

Connection（连接）

AUTH

AUTH password

通过设置配置文件中 `requirepass` 项的值(使用命令 `CONFIG SET requirepass password`)，可以使用密码来保护 Redis 服务器。

如果开启了密码保护的话，在每次连接 Redis 服务器之后，就要使用 `AUTH` 命令解锁，解锁之后才能使用其他 Redis 命令。

如果 `AUTH` 命令给定的密码 `password` 和配置文件中的密码相符的话，服务器会返回 `OK` 并开始接受命令输入。

另一方面，假如密码不匹配的话，服务器将返回一个错误，并要求客户端需重新输入密码。

Warning

因为 Redis 高性能的特点，在很短时间内尝试猜测非常多个密码是有可能的，因此请确保使用的密码足够复杂和足够长，以免遭受密码猜测攻击。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

密码匹配时返回 `OK` ，否则返回一个错误。

```
# 设置密码

redis> CONFIG SET requirepass secret_password   # 将密码设置为 secret
OK

redis> QUIT                                      # 退出再连接，让新密码

[huangz@mypad]$ redis

redis> PING                                      # 未验证密码，操作被拒
(error) ERR operation not permitted

redis> AUTH wrong_password_testing               # 尝试输入错误的密码
(error) ERR invalid password

redis> AUTH secret_password                      # 输入正确的密码
OK

redis> PING                                      # 密码验证成功，可以正
PONG

# 清空密码

redis> CONFIG SET requirepass ""   # 通过将密码设为空字符来清空密码
OK

redis> QUIT

$ redis                            # 重新进入客户端

redis> PING                        # 执行命令不再需要密码，清空密码操作成
PONG
```

ECHO

ECHO message

打印一个特定的信息 `message` ，测试时使用。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

`message` 自身。

```
redis> ECHO "Hello Moto"
"Hello Moto"

redis> ECHO "Goodbye Moto"
"Goodbye Moto"
```

PING

PING

使用客户端向 Redis 服务器发送一个 `PING` ，如果服务器运作正常的话，会返回一个 `PONG` 。

通常用于测试与服务器的连接是否仍然生效，或者用于测量延迟值。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

如果连接正常就返回一个 `PONG` ，否则返回一个连接错误。

```
# 客户端和服务器连接正常

redis> PING
PONG

# 客户端和服务器连接不正常(网络不正常或服务器未能正常运行)

redis 127.0.0.1:6379> PING
Could not connect to Redis at 127.0.0.1:6379: Connection refused
```

QUIT

QUIT

请求服务器关闭与当前客户端的连接。

一旦所有等待中的回复(如果有的话)顺利写入到客户端，连接就会被关闭。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

总是返回 `OK` (但是不会被打印显示，因为当时 Redis-cli 已经退出)。

```
$ redis

redis> QUIT

$
```

SELECT

SELECT index

切换到指定的数据库，数据库索引号 `index` 用数字值指定，以 `0` 作为起始索引值。

默认使用 `0` 号数据库。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

`OK`

```
redis> SET db_number 0         # 默认使用 0 号数据库
OK

redis> SELECT 1                # 使用 1 号数据库
OK

redis[1]> GET db_number        # 已经切换到 1 号数据库，注意 Redis 现在
(nil)

redis[1]> SET db_number 1
OK

redis[1]> GET db_number
"1"

redis[1]> SELECT 3             # 再切换到 3 号数据库
OK

redis[3]>                      # 提示符从 [1] 改变成了 [3]
```

Server（服务器）

BGREWRITEAOF

BGREWRITEAOF

执行一个 AOF文件 重写操作。重写会创建一个当前 AOF 文件的体积优化版本。

即使 BGREWRITEAOF 执行失败，也不会有任何数据丢失，因为旧的 AOF 文件在 BGREWRITEAOF 成功之前不会被修改。

重写操作只会在没有其他持久化工作在后台执行时被触发，也就是说：

- 如果 Redis 的子进程正在执行快照的保存工作，那么 AOF 重写的操作会被预定(scheduled)，等到保存工作完成之后再执行 AOF 重写。在这种情况下，BGREWRITEAOF 的返回值仍然是 `OK` ，但还会加上一条额外的信息，说明 BGREWRITEAOF 要等到保存操作完成之后才能执行。在 Redis 2.6 或以上的版本，可以使用 *INFO* 命令查看 BGREWRITEAOF 是否被预定。
- 如果已经有别的 AOF 文件重写在执行，那么 BGREWRITEAOF 返回一个错误，并且这个新的 BGREWRITEAOF 请求也不会被预定到下次执行。

从 Redis 2.4 开始， AOF 重写由 Redis 自行触发， BGREWRITEAOF 仅仅用于手动触发重写操作。

请移步 持久化文档(英文) 查看更多相关细节。

可用版本：

>= 1.0.0

时间复杂度：

O(N)， `N` 为要追加到 AOF 文件中的数据数量。

返回值：

反馈信息。

```
redis> BGREWRITEAOF
Background append only file rewriting started
```

BGSAVE

在后台异步(Asynchronously)保存当前数据库的数据到磁盘。

BGSAVE 命令执行之后立即返回 `OK` ，然后 Redis fork 出一个新子进程，原来的 Redis 进程(父进程)继续处理客户端请求，而子进程则负责将数据保存到磁盘，然后退出。

客户端可以通过 *LASTSAVE* 命令查看相关信息，判断 BGSAVE 命令是否执行成功。

请移步 持久化文档 查看更多相关细节。

可用版本：

>= 1.0.0

时间复杂度：

O(N)， `N` 为要保存到数据库中的 key 的数量。

返回值：

反馈信息。

```
redis> BGSAVE
Background saving started
```

CLIENT GETNAME

CLIENT GETNAME

返回 *CLIENT SETNAME* 命令为连接设置的名字。

因为新创建的连接默认是没有名字的， 对于没有名字的连接， *CLIENT GETNAME* 返回空白回复。

可用版本

>= 2.6.9

时间复杂度

O(1)

返回值

如果连接没有设置名字，那么返回空白回复；如果有设置名字，那么返回名字。

```
# 新连接默认没有名字

redis 127.0.0.1:6379> CLIENT GETNAME
(nil)

# 设置名字

redis 127.0.0.1:6379> CLIENT SETNAME hello-world-connection
OK

# 返回名字

redis 127.0.0.1:6379> CLIENT GETNAME
"hello-world-connection"
```

CLIENT KILL

CLIENT KILL ip:port

关闭地址为 `ip:port` 的客户端。

`ip:port` 应该和 *CLIENT LIST* 命令输出的其中一行匹配。

因为 Redis 使用单线程设计，所以当 Redis 正在执行命令的时候，不会有客户端被断开连接。

如果要被断开连接的客户端正在执行命令，那么当这个命令执行之后，在发送下一个命令的时候，它就会收到一个网络错误，告知它自身的连接已被关闭。

可用版本

>= 2.4.0

时间复杂度

O(N)， N 为已连接的客户端数量。

返回值

当指定的客户端存在，且被成功关闭时，返回 `OK` 。

```
# 列出所有已连接客户端

redis 127.0.0.1:6379> CLIENT LIST
addr=127.0.0.1:43501 fd=5 age=10 idle=0 flags=N db=0 sub=0 psub=0 r

# 杀死当前客户端的连接

redis 127.0.0.1:6379> CLIENT KILL 127.0.0.1:43501
OK

# 之前的连接已经被关闭，CLI 客户端又重新建立了连接
# 之前的端口是 43501 ，现在是 43504

redis 127.0.0.1:6379> CLIENT LIST
addr=127.0.0.1:43504 fd=5 age=0 idle=0 flags=N db=0 sub=0 psub=0 mu
```

CLIENT LIST

CLIENT LIST

以人类可读的格式，返回所有连接到服务器的客户端信息和统计数据。

```
redis> CLIENT LIST
addr=127.0.0.1:43143 fd=6 age=183 idle=0 flags=N db=0 sub=0 psub=0 
addr=127.0.0.1:43163 fd=5 age=35 idle=15 flags=N db=0 sub=0 psub=0 
addr=127.0.0.1:43167 fd=7 age=24 idle=6 flags=N db=0 sub=0 psub=0 r
```

可用版本

>= 2.4.0

时间复杂度

O(N)， N 为连接到服务器的客户端数量。

返回值

命令返回多行字符串，这些字符串按以下形式被格式化：

- 每个已连接客户端对应一行（以 `LF` 分割）
- 每行字符串由一系列 `属性=值` 形式的域组成，每个域之间以空格分开

以下是域的含义：

- `addr` ：客户端的地址和端口
- `fd` ：套接字所使用的文件描述符
- `age` ：以秒计算的已连接时长
- `idle` ：以秒计算的空闲时长
- `flags` ：客户端 flag （见下文）
- `db` ：该客户端正在使用的数据库 ID
- `sub` ：已订阅频道的数量
- `psub` ：已订阅模式的数量
- `multi` ：在事务中被执行的命令数量
- `qbuf` ：查询缓冲区的长度（字节为单位， `0` 表示没有分配查询缓冲区）
- `qbuf-free` ：查询缓冲区剩余空间的长度（字节为单位， `0` 表示没有剩余空间）
- `obl` ：输出缓冲区的长度（字节为单位， `0` 表示没有分配输出缓冲区）
- `oll` ：输出列表包含的对象数量（当输出缓冲区没有剩余空间时，命令回复会以字符串对象的形式被入队到这个队列里）
- `omem` ：输出缓冲区和输出列表占用的内存总量
- `events` ：文件描述符事件（见下文）
- `cmd` ：最近一次执行的命令

客户端 flag 可以由以下部分组成：

- `O` ：客户端是 MONITOR 模式下的附属节点（slave）
- `S` ：客户端是一般模式下（normal）的附属节点
- `M` ：客户端是主节点（master）
- `x` ：客户端正在执行事务
- `b` ：客户端正在等待阻塞事件
- `i` ：客户端正在等待 VM I/O 操作（已废弃）
- `d` ：一个受监视（watched）的键已被修改， `EXEC` 命令将失败
- `c` : 在将回复完整地写出之后，关闭链接
- `u` : 客户端未被阻塞（unblocked）
- `A` : 尽可能快地关闭连接
- `N` : 未设置任何 flag

文件描述符事件可以是：

- `r` : 客户端套接字（在事件 loop 中）是可读的（readable）
- `w` : 客户端套接字（在事件 loop 中）是可写的（writeable）

Note

为了 debug 的需要，经常会对域进行添加和删除，一个安全的 Redis 客户端应该可以对 `CLIENT LIST` 的输出进行相应的处理（parse），比如忽略不存在的域，跳过未知域，诸如此类。

CLIENT SETNAME

CLIENT SETNAME connection-name

为当前连接分配一个名字。

这个名字会显示在 *CLIENT LIST* 命令的结果中， 用于识别当前正在与服务器进行连接的客户端。

举个例子， 在使用 Redis 构建队列（queue）时， 可以根据连接负责的任务（role）， 为信息生产者（producer）和信息消费者（consumer）分别设置不同的名字。

名字使用 Redis 的字符串类型来保存， 最大可以占用 512 MB 。 另外， 为了避免和 *CLIENT LIST* 命令的输出格式发生冲突， 名字里不允许使用空格。

要移除一个连接的名字， 可以将连接的名字设为空字符串 `""` 。

使用 *CLIENT GETNAME* 命令可以取出连接的名字。

新创建的连接默认是没有名字的。

Tip

在 Redis 应用程序发生连接泄漏时，为连接设置名字是一种很好的 debug 手段。

可用版本

>= 2.6.9

时间复杂度

O(1)

返回值

设置成功时返回 `OK` 。

```
# 新连接默认没有名字

redis 127.0.0.1:6379> CLIENT GETNAME
(nil)

# 设置名字

redis 127.0.0.1:6379> CLIENT SETNAME hello-world-connection
OK

# 返回名字

redis 127.0.0.1:6379> CLIENT GETNAME
"hello-world-connection"

# 在客户端列表中查看

redis 127.0.0.1:6379> CLIENT LIST
addr=127.0.0.1:36851
fd=5
name=hello-world-connection     # <- 名字
age=51
...

# 清除名字

redis 127.0.0.1:6379> CLIENT SETNAME        # 只用空格是不行的！
(error) ERR Syntax error, try CLIENT (LIST | KILL ip:port)

redis 127.0.0.1:6379> CLIENT SETNAME ""     # 必须双引号显示包围
OK

redis 127.0.0.1:6379> CLIENT GETNAME        # 清除完毕
(nil)
```

CONFIG GET

CONFIG GET parameter

CONFIG GET 命令用于取得运行中的 Redis 服务器的配置参数(configuration parameters)，在 Redis 2.4 版本中， 有部分参数没有办法用 `CONFIG GET` 访问，但是在最新的 Redis 2.6 版本中，所有配置参数都已经可以用 `CONFIG GET` 访问了。

CONFIG GET 接受单个参数 `parameter` 作为搜索关键字，查找所有匹配的配置参数，其中参数和值以“键-值对”(key-value pairs)的方式排列。

比如执行 `CONFIG GET s*` 命令，服务器就会返回所有以 `s` 开头的配置参数及参数的值：

```
redis> CONFIG GET s*
1) "save"                        # 参数名：save
2) "900 1 300 10 60 10000"       # save 参数的值
3) "slave-serve-stale-data"      # 参数名： slave-serve-stale-data
4) "yes"                         # slave-serve-stale-data 参数的值
5) "set-max-intset-entries"      # ...
6) "512"
7) "slowlog-log-slower-than"
8) "1000"
9) "slowlog-max-len"
10) "1000"
```

如果你只是寻找特定的某个参数的话，你当然也可以直接指定参数的名字：

```
redis> CONFIG GET slowlog-max-len
1) "slowlog-max-len"
2) "1000"
```

使用命令 `CONFIG GET *` ，可以列出 `CONFIG GET` 命令支持的所有参数：

```
redis> CONFIG GET *
1) "dir"
2) "/var/lib/redis"
3) "dbfilename"
4) "dump.rdb"
5) "requirepass"
6) (nil)
7) "masterauth"
8) (nil)
9) "maxmemory"
10) "0"
11) "maxmemory-policy"
12) "volatile-lru"
13) "maxmemory-samples"
14) "3"
15) "timeout"
16) "0"
17) "appendonly"
18) "no"
# ...
49) "loglevel"
50) "verbose"
```

所有被 CONFIG SET 所支持的配置参数都可以在配置文件 redis.conf 中找到，不过 CONFIG GET 和 CONFIG SET 使用的格式和 redis.conf 文件所使用的格式有以下两点不同：

- 10kb 、 2gb 这些在配置文件中所使用的储存单位缩写，不可以用在 CONFIG 命令中， CONFIG SET 的值只能通过数字值显式地设定。像 CONFIG SET xxx 1k 这样的命令是错误的，正确的格式是 CONFIG SET xxx 1000 。
- save 选项在 redis.conf 中是用多行文字储存的，但在 CONFIG GET 命令中，它只打印一行文字。以下是 save 选项在 redis.conf 文件中的表示： save 900 1``save 300 10``save 60 10000 但是 CONFIG GET 命令的输出只有一行： redis> CONFIG GET save``1) "save"``2) "900 1 300 10 60 10
上面 save 参数的三个值表示：在 900 秒内最少有 1 个 key 被改动，或者 300 秒内最少有 10 个 key 被改动，又或者 60 秒内最少有 1000 个 key 被改动，以上三个条件随便满足一个，就触发一次保存操作。

可用版本：

>= 2.0.0

时间复杂度：

不明确

返回值：

给定配置参数的值。

CONFIG RESETSTAT

CONFIG RESETSTAT

重置 *INFO* 命令中的某些统计数据，包括：

- Keyspace hits (键空间命中次数)
- Keyspace misses (键空间不命中次数)
- Number of commands processed (执行命令的次数)
- Number of connections received (连接服务器的次数)
- Number of expired keys (过期key的数量)
- Number of rejected connections (被拒绝的连接数量)
- Latest fork(2) time(最后执行 fork(2) 的时间)
- The `aof_delayed_fsync` counter(`aof_delayed_fsync` 计数器的值)

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

总是返回 `OK` 。

```
# 重置前

redis 127.0.0.1:6379> INFO
# Server
redis_version:2.5.3
redis_git_sha1:d0407c2d
redis_git_dirty:0
arch_bits:32
multiplexing_api:epoll
gcc_version:4.6.3
process_id:11095
run_id:ef1f6b6c7392e52d6001eaf777acbe547d1192e2
tcp_port:6379
uptime_in_seconds:6
uptime_in_days:0
lru_clock:1205426

# Clients
connected_clients:1
client_longest_output_list:0
client_biggest_input_buf:0
blocked_clients:0
```

```
# Memory
used_memory:331076
used_memory_human:323.32K
used_memory_rss:1568768
used_memory_peak:293424
used_memory_peak_human:286.55K
used_memory_lua:16384
mem_fragmentation_ratio:4.74
mem_allocator:jemalloc-2.2.5

# Persistence
loading:0
aof_enabled:0
changes_since_last_save:0
bgsave_in_progress:0
last_save_time:1333260015
last_bgsave_status:ok
bgrewriteaof_in_progress:0

# Stats
total_connections_received:1
total_commands_processed:0
instantaneous_ops_per_sec:0
rejected_connections:0
expired_keys:0
evicted_keys:0
keyspace_hits:0
keyspace_misses:0
pubsub_channels:0
pubsub_patterns:0
latest_fork_usec:0

# Replication
role:master
connected_slaves:0

# CPU
used_cpu_sys:0.01
used_cpu_user:0.00
used_cpu_sys_children:0.00
used_cpu_user_children:0.00

# Keyspace
db0:keys=20,expires=0

# 重置

redis 127.0.0.1:6379> CONFIG RESETSTAT
OK

# 重置后

redis 127.0.0.1:6379> INFO
```

```
# Server
redis_version:2.5.3
redis_git_sha1:d0407c2d
redis_git_dirty:0
arch_bits:32
multiplexing_api:epoll
gcc_version:4.6.3
process_id:11095
run_id:ef1f6b6c7392e52d6001eaf777acbe547d1192e2
tcp_port:6379
uptime_in_seconds:134
uptime_in_days:0
lru_clock:1205438

# Clients
connected_clients:1
client_longest_output_list:0
client_biggest_input_buf:0
blocked_clients:0

# Memory
used_memory:331076
used_memory_human:323.32K
used_memory_rss:1568768
used_memory_peak:330280
used_memory_peak_human:322.54K
used_memory_lua:16384
mem_fragmentation_ratio:4.74
mem_allocator:jemalloc-2.2.5

# Persistence
loading:0
aof_enabled:0
changes_since_last_save:0
bgsave_in_progress:0
last_save_time:1333260015
last_bgsave_status:ok
bgrewriteaof_in_progress:0

# Stats
total_connections_received:0
total_commands_processed:1
instantaneous_ops_per_sec:0
rejected_connections:0
expired_keys:0
evicted_keys:0
keyspace_hits:0
keyspace_misses:0
pubsub_channels:0
pubsub_patterns:0
latest_fork_usec:0

# Replication
```

```
role:master
connected_slaves:0

# CPU
used_cpu_sys:0.05
used_cpu_user:0.02
used_cpu_sys_children:0.00
used_cpu_user_children:0.00

# Keyspace
db0:keys=20,expires=0
```

CONFIG REWRITE

CONFIG REWRITE

CONFIG REWRITE 命令对启动 Redis 服务器时所指定的 `redis.conf` 文件进行改写： 因为 *CONFIG SET* 命令可以对服务器的当前配置进行修改， 而修改后的配置可能和 `redis.conf` 文件中所描述的配置不一样， *CONFIG REWRITE* 的作用就是通过尽可能少的修改， 将服务器当前所使用的配置记录到 `redis.conf` 文件中。

重写会以非常保守的方式进行：

- 原有 `redis.conf` 文件的整体结构和注释会被尽可能地保留。
- 如果一个选项已经存在于原有 `redis.conf` 文件中， 那么对该选项的重写会在选项原本所在的位置（行号）上进行。
- 如果一个选项不存在于原有 `redis.conf` 文件中， 并且该选项被设置为默认值， 那么重写程序不会将这个选项添加到重写后的 `redis.conf` 文件中。
- 如果一个选项不存在于原有 `redis.conf` 文件中， 并且该选项被设置为非默认值， 那么这个选项将被添加到重写后的 `redis.conf` 文件的末尾。
- 未使用的行会被留白。 比如说， 如果你在原有 `redis.conf` 文件上设置了数个关于 `save` 选项的参数， 但现在你将这些 `save` 参数的一个或全部都关闭了， 那么这些不再使用的参数原本所在的行就会变成空白的。

即使启动服务器时所指定的 `redis.conf` 文件已经不再存在， *CONFIG REWRITE* 命令也可以重新构建并生成出一个新的 `redis.conf` 文件。

另一方面， 如果启动服务器时没有载入 `redis.conf` 文件， 那么执行 *CONFIG REWRITE* 命令将引发一个错误。

原子性重写

对 `redis.conf` 文件的重写是原子性的， 并且是一致的： 如果重写出错或重写期间服务器崩溃， 那么重写失败， 原有 `redis.conf` 文件不会被修改。 如果重写成功， 那么 `redis.conf` 文件为重写后的新文件。

可用版本

>= 2.8.0

返回值

一个状态值：如果配置重写成功则返回 `OK` ， 失败则返回一个错误。

测试

以下是执行 *CONFIG REWRITE* 前，被载入到 Redis 服务器的 `redis.conf` 文件中关于 `appendonly` 选项的设置：

```
# ... 其他选项

appendonly no

# ... 其他选项
```

在执行以下命令之后：

```
127.0.0.1:6379> CONFIG GET appendonly           # appendonly 处于关
1) "appendonly"
2) "no"

127.0.0.1:6379> CONFIG SET appendonly yes       # 打开 appendonly
OK

127.0.0.1:6379> CONFIG GET appendonly
1) "appendonly"
2) "yes"

127.0.0.1:6379> CONFIG REWRITE                  # 将 appendonly 的修
OK
```

重写后的 `redis.conf` 文件中的 `appendonly` 选项将被改写：

```
# ... 其他选项

appendonly yes

# ... 其他选项
```

CONFIG SET

CONFIG SET parameter value

CONFIG SET 命令可以动态地调整 Redis 服务器的配置(configuration)而无须重启。

你可以使用它修改配置参数，或者改变 Redis 的持久化(Persistence)方式。

CONFIG SET 可以修改的配置参数可以使用命令 `CONFIG GET *` 来列出，所有被 CONFIG SET 修改的配置参数都会立即生效。

关于 CONFIG SET 命令的更多消息，请参见命令 *CONFIG GET* 的说明。

关于如何使用 CONFIG SET 命令修改 Redis 持久化方式，请参见 Redis Persistence 。

可用版本：

>= 2.0.0

时间复杂度：

不明确

返回值：

当设置成功时返回 `OK` ，否则返回一个错误。

```
redis> CONFIG GET slowlog-max-len
1) "slowlog-max-len"
2) "1024"

redis> CONFIG SET slowlog-max-len 10086
OK

redis> CONFIG GET slowlog-max-len
1) "slowlog-max-len"
2) "10086"
```

DBSIZE

DBSIZE

返回当前数据库的 key 的数量。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

当前数据库的 key 的数量。

```
redis> DBSIZE
(integer) 5

redis> SET new_key "hello_moto"     # 增加一个 key 试试
OK

redis> DBSIZE
(integer) 6
```

DEBUG OBJECT

DEBUG OBJECT key

DEBUG OBJECT 是一个调试命令，它不应被客户端所使用。

查看 *OBJECT* 命令获取更多信息。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

当 `key` 存在时，返回有关信息。当 `key` 不存在时，返回一个错误。

```
redis> DEBUG OBJECT my_pc
Value at:0xb6838d20 refcount:1 encoding:raw serializedlength:9 lru

redis> DEBUG OBJECT your_mac
(error) ERR no such key
```

DEBUG SEGFAULT

DEBUG SEGFAULT

执行一个不合法的内存访问从而让 Redis 崩溃，仅在开发时用于 BUG 模拟。

可用版本：

>= 1.0.0

时间复杂度：

不明确

返回值：

无

```
redis> DEBUG SEGFAULT
Could not connect to Redis at: Connection refused

not connected>
```

FLUSHALL

FLUSHALL

清空整个 Redis 服务器的数据(删除所有数据库的所有 key)。

此命令从不失败。

可用版本：

>= 1.0.0

时间复杂度：

尚未明确

返回值：

总是返回 `OK` 。

```
redis> DBSIZE               # 0 号数据库的 key 数量
(integer) 9

redis> SELECT 1             # 切换到 1 号数据库
OK

redis[1]> DBSIZE            # 1 号数据库的 key 数量
(integer) 6

redis[1]> flushall          # 清空所有数据库的所有 key
OK

redis[1]> DBSIZE            # 不但 1 号数据库被清空了
(integer) 0

redis[1]> SELECT 0          # 0 号数据库(以及其他所有数据库)也一样
OK

redis> DBSIZE
(integer) 0
```

FLUSHDB

FLUSHDB

清空当前数据库中的所有 key。

此命令从不失败。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

总是返回 `OK` 。

```
redis> DBSIZE    # 清空前的 key 数量
(integer) 4

redis> FLUSHDB
OK

redis> DBSIZE    # 清空后的 key 数量
(integer) 0
```

INFO

INFO [section]

以一种易于解释（parse）且易于阅读的格式，返回关于 Redis 服务器的各种信息和统计数值。

通过给定可选的参数 `section` ，可以让命令只返回某一部分的信息：

- `server` : 一般 Redis 服务器信息，包含以下域：

  > *`redis_version` : Redis* 服务器版本 > `redis_git_sha1` : Git SHA1 > *`redis_git_dirty` : Git dirty flag* > `os` : Redis 服务器的宿主操作系统 > *`arch_bits` :* 架构（*32* 或 *64* 位） > `multiplexing_api` : Redis 所使用的事件处理机制 > *`gcc_version` :* 编译 *Redis* 时所使用的 *GCC* 版本 > `process_id` : 服务器进程的 PID > *`run_id` : Redis* 服务器的随机标识符（用于 *Sentinel* 和集群） > `tcp_port` : TCP/IP 监听端口 > *`uptime_in_seconds` :* 自 *Redis* 服务器启动以来，经过的秒数 > `uptime_in_days` : 自 Redis 服务器启动以来，经过的天数 > * `lru_clock` : 以分钟为单位进行自增的时钟，用于 LRU 管理

- `clients` : 已连接客户端信息，包含以下域：

  > *`connected_clients` :* 已连接客户端的数量（不包括通过从属服务器连接的客户端） > `client_longest_output_list` : 当前连接的客户端当中，最长的输出列表 > *`client_longest_input_buf` :* 当前连接的客户端当中，最大输入缓存 > `blocked_clients` : 正在等待阻塞命令（BLPOP、BRPOP、BRPOPLPUSH）的客户端的数量

- `memory` : 内存信息，包含以下域：

  > *`used_memory` :* 由 *Redis* 分配器分配的内存总量，以字节（*byte*）为单位 > `used_memory_human` : 以人类可读的格式返回 Redis 分配的内存总量 > *`used_memory_rss` :* 从操作系统的角度，返回 *Redis* 已分配的内存总量（俗称常驻集大小）。这个值和 *`top`* 、 *`ps`* 等命令的输出一致。 > `used_memory_peak` : Redis 的内存消耗峰值（以字节为单位） > *`used_memory_peak_human` :* 以人类可读的格式返回 *Redis* 的内存消耗峰值 > `used_memory_lua` : Lua 引擎所使用的内存大小（以字节为单位） > *`mem_fragmentation_ratio` : `used_memory_rss`* 和 *`used_memory`* 之间的比率 > `mem_allocator` : 在编译时指定的， Redis 所使用的内存分配器。可以是 libc 、 jemalloc 或者 tcmalloc 。 > > 在理想情况下， `used_memory_rss` 的值应该只比 `used_memory` 稍微高一点儿。当 `rss > used` ，且两者的值相差较大时，表示存在（内部或外部的）内存碎片。内存碎片的比率可以通过 `mem_fragmentation_ratio` 的值看出。当 `used > rss` 时，表示 Redis 的部分内存被操作系统换出到交换空间了，在这种情况下，操作可能会产生明显的延迟。 > > Because Redis does not have control over how its allocations are mapped to memory pages, high

`used_memory_rss` is often the result of a spike in memory usage. > > 当 Redis 释放内存时，分配器可能会，也可能不会，将内存返还给操作系统。如果 Redis 释放了内存，却没有将内存返还给操作系统，那么 `used_memory` 的值可能和操作系统显示的 Redis 内存占用并不一致。查看 `used_memory_peak` 的值可以验证这种情况是否发生。

- `persistence` : `RDB` 和 `AOF` 的相关信息
- `stats` : 一般统计信息
- `replication` : 主/从复制信息
- `cpu` : CPU 计算量统计信息
- `commandstats` : Redis 命令统计信息
- `cluster` : Redis 集群信息
- `keyspace` : 数据库相关的统计信息

除上面给出的这些值以外，参数还可以是下面这两个：

- `all` : 返回所有信息
- `default` : 返回默认选择的信息

当不带参数直接调用 INFO 命令时，使用 `default` 作为默认参数。

Note

不同版本的 Redis 可能对返回的一些域进行了增加或删减。

因此，一个健壮的客户端程序在对 *INFO* 命令的输出进行分析时，应该能够跳过不认识的域，并且妥善地处理丢失不见的域。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

具体请参见下面的测试代码。

```
redis> INFO
# Server
redis_version:2.5.9
redis_git_sha1:473f3090
redis_git_dirty:0
os:Linux 3.3.7-1-ARCH i686
arch_bits:32
multiplexing_api:epoll
```

```
gcc_version:4.7.0
process_id:8104
run_id:bc9e20c6f0aac67d0d396ab950940ae4d1479ad1
tcp_port:6379
uptime_in_seconds:7
uptime_in_days:0
lru_clock:1680564

# Clients
connected_clients:1
client_longest_output_list:0
client_biggest_input_buf:0
blocked_clients:0

# Memory
used_memory:439304
used_memory_human:429.01K
used_memory_rss:13897728
used_memory_peak:401776
used_memory_peak_human:392.36K
used_memory_lua:20480
mem_fragmentation_ratio:31.64
mem_allocator:jemalloc-3.0.0

# Persistence
loading:0
rdb_changes_since_last_save:0
rdb_bgsave_in_progress:0
rdb_last_save_time:1338011402
rdb_last_bgsave_status:ok
rdb_last_bgsave_time_sec:-1
rdb_current_bgsave_time_sec:-1
aof_enabled:0
aof_rewrite_in_progress:0
aof_rewrite_scheduled:0
aof_last_rewrite_time_sec:-1
aof_current_rewrite_time_sec:-1

# Stats
total_connections_received:1
total_commands_processed:0
instantaneous_ops_per_sec:0
rejected_connections:0
expired_keys:0
evicted_keys:0
keyspace_hits:0
keyspace_misses:0
pubsub_channels:0
pubsub_patterns:0
latest_fork_usec:0

# Replication
role:master
```

```
connected_slaves:0

# CPU
used_cpu_sys:0.03
used_cpu_user:0.01
used_cpu_sys_children:0.00
used_cpu_user_children:0.00

# Keyspace
```

LASTSAVE

LASTSAVE

返回最近一次 Redis 成功将数据保存到磁盘上的时间，以 UNIX 时间戳格式表示。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

一个 UNIX 时间戳。

```
redis> LASTSAVE
(integer) 1324043588
```

MONITOR

MONITOR

实时打印出 Redis 服务器接收到的命令，调试用。

可用版本：

>= 1.0.0

时间复杂度：

不明确

返回值：

总是返回 `OK` 。

```
127.0.0.1:6379> MONITOR
OK
# 以第一个打印值为例
# 1378822099.421623 是时间戳
# [0 127.0.0.1:56604] 中的 0 是数据库号码， 127... 是 IP 地址和端口
# "PING" 是被执行的命令
1378822099.421623 [0 127.0.0.1:56604] "PING"
1378822105.089572 [0 127.0.0.1:56604] "SET" "msg" "hello world"
1378822109.036925 [0 127.0.0.1:56604] "SET" "number" "123"
1378822140.649496 [0 127.0.0.1:56604] "SADD" "fruits" "Apple" "Bana
1378822154.117160 [0 127.0.0.1:56604] "EXPIRE" "msg" "10086"
1378822257.329412 [0 127.0.0.1:56604] "KEYS" "*"
1378822258.690131 [0 127.0.0.1:56604] "DBSIZE"
```

PSYNC

PSYNC <MASTER_RUN_ID> <OFFSET>

用于复制功能(replication)的内部命令。

更多信息请参考 复制（*Replication*） 文档。

可用版本：

>= 2.8.0

时间复杂度：

不明确

返回值：

不明确

```
127.0.0.1:6379> PSYNC ? -1
"REDIS0006\xfe\x00\x00\x02kk\x02vv\x00\x03msg\x05hello\xff\xc3\x96F
```

SAVE

SAVE

SAVE 命令执行一个同步保存操作，将当前 Redis 实例的所有数据快照(snapshot)以 RDB 文件的形式保存到硬盘。

一般来说，在生产环境很少执行 SAVE 操作，因为它会阻塞所有客户端，保存数据库的任务通常由 *BGSAVE* 命令异步地执行。然而，如果负责保存数据的后台子进程不幸出现问题时， SAVE 可以作为保存数据的最后手段来使用。

请参考文档： Redis 的持久化运作方式(英文) 以获取更多消息。

可用版本：

>= 1.0.0

时间复杂度：

O(N)， `N` 为要保存到数据库中的 key 的数量。

返回值：

保存成功时返回 `OK` 。

```
redis> SAVE
OK
```

SHUTDOWN

SHUTDOWN

SHUTDOWN 命令执行以下操作：

- 停止所有客户端
- 如果有至少一个保存点在等待，执行 *SAVE* 命令
- 如果 AOF 选项被打开，更新 AOF 文件
- 关闭 redis 服务器(server)

如果持久化被打开的话， SHUTDOWN 命令会保证服务器正常关闭而不丢失任何数据。

另一方面，假如只是单纯地执行 *SAVE* 命令，然后再执行 *QUIT* 命令，则没有这一保证 —— 因为在执行 *SAVE* 之后、执行 *QUIT* 之前的这段时间中间，其他客户端可能正在和服务器进行通讯，这时如果执行 *QUIT* 就会造成数据丢失。

SAVE 和 NOSAVE 修饰符

通过使用可选的修饰符，可以修改 SHUTDOWN 命令的表现。比如说：

- 执行 `SHUTDOWN SAVE` 会强制让数据库执行保存操作，即使没有设定(configure)保存点
- 执行 `SHUTDOWN NOSAVE` 会阻止数据库执行保存操作，即使已经设定有一个或多个保存点(你可以将这一用法看作是强制停止服务器的一个假想的 ABORT 命令)

可用版本：

>= 1.0.0

时间复杂度：

不明确

返回值：

执行失败时返回错误。执行成功时不返回任何信息，服务器和客户端的连接断开，客户端自动退出。

```
redis> PING
PONG

redis> SHUTDOWN

$

$ redis
Could not connect to Redis at: Connection refused
not connected>
```

SLAVEOF

`SLAVEOF host port`

SLAVEOF 命令用于在 Redis 运行时动态地修改复制(replication)功能的行为。

通过执行 `SLAVEOF host port` 命令，可以将当前服务器转变为指定服务器的从属服务器(slave server)。

如果当前服务器已经是某个主服务器(master server)的从属服务器，那么执行 `SLAVEOF host port` 将使当前服务器停止对旧主服务器的同步，丢弃旧数据集，转而开始对新主服务器进行同步。

另外，对一个从属服务器执行命令 `SLAVEOF NO ONE` 将使得这个从属服务器关闭复制功能，并从从属服务器转变回主服务器，原来同步所得的数据集不会被丢弃。

利用『 `SLAVEOF NO ONE` 不会丢弃同步所得数据集』这个特性，可以在主服务器失败的时候，将从属服务器用作新的主服务器，从而实现无间断运行。

可用版本：

>= 1.0.0

时间复杂度：

`SLAVEOF host port` ，O(N)， `N` 为要同步的数据数量。 `SLAVEOF NO ONE` ， O(1) 。

返回值：

总是返回 `OK` 。

```
redis> SLAVEOF 127.0.0.1 6379
OK

redis> SLAVEOF NO ONE
OK
```

SLOWLOG

SLOWLOG subcommand [argument]

什么是 **SLOWLOG**

Slow log 是 Redis 用来记录查询执行时间的日志系统。

查询执行时间指的是不包括像客户端响应(talking)、发送回复等 IO 操作，而单单是执行一个查询命令所耗费的时间。

另外，slow log 保存在内存里面，读写速度非常快，因此你可以放心地使用它，不必担心因为开启 slow log 而损害 Redis 的速度。

设置 **SLOWLOG**

Slow log 的行为由两个配置参数(configuration parameter)指定，可以通过改写 redis.conf 文件或者用 `CONFIG GET` 和 `CONFIG SET` 命令对它们动态地进行修改。

第一个选项是 `slowlog-log-slower-than` ，它决定要对执行时间大于多少微秒(microsecond，1秒 = 1,000,000 微秒)的查询进行记录。

比如执行以下命令将让 slow log 记录所有查询时间大于等于 100 微秒的查询：

```
CONFIG SET slowlog-log-slower-than 100
```

而以下命令记录所有查询时间大于 1000 微秒的查询：

```
CONFIG SET slowlog-log-slower-than 1000
```

另一个选项是 `slowlog-max-len` ，它决定 slow log 最多能保存多少条日志，slow log 本身是一个 FIFO 队列，当队列大小超过 `slowlog-max-len` 时，最旧的一条日志将被删除，而最新的一条日志加入到 slow log ，以此类推。

以下命令让 slow log 最多保存 1000 条日志：

```
CONFIG SET slowlog-max-len 1000
```

使用 `CONFIG GET` 命令可以查询两个选项的当前值：

```
redis> CONFIG GET slowlog-log-slower-than
1) "slowlog-log-slower-than"
2) "1000"

redis> CONFIG GET slowlog-max-len
1) "slowlog-max-len"
2) "1000"
```

查看 **slow log**

要查看 slow log ，可以使用 `SLOWLOG GET` 或者 `SLOWLOG GET number` 命令，前者打印所有 slow log ，最大长度取决于 `slowlog-max-len` 选项的值，而 `SLOWLOG GET number` 则只打印指定数量的日志。

最新的日志会最先被打印：

```
# 为测试需要，将 slowlog-log-slower-than 设成了 10 微秒

redis> SLOWLOG GET
1) 1) (integer) 12                      # 唯一性(unique)的日志标识符
   2) (integer) 1324097834              # 被记录命令的执行时间点，以 UN
   3) (integer) 16                      # 查询执行时间，以微秒为单位
   4) 1) "CONFIG"                       # 执行的命令，以数组的形式排列
      2) "GET"                          # 这里完整的命令是 CONFIG GET
      3) "slowlog-log-slower-than"

2) 1) (integer) 11
   2) (integer) 1324097825
   3) (integer) 42
   4) 1) "CONFIG"
      2) "GET"
      3) "*"

3) 1) (integer) 10
   2) (integer) 1324097820
   3) (integer) 11
   4) 1) "CONFIG"
      2) "GET"
      3) "slowlog-log-slower-than"

# ...
```

日志的唯一 id 只有在 Redis 服务器重启的时候才会重置，这样可以避免对日志的重复处理(比如你可能会想在每次发现新的慢查询时发邮件通知你)。

查看当前日志的数量

使用命令 `SLOWLOG LEN` 可以查看当前日志的数量。

请注意这个值和 `slower-max-len` 的区别，它们一个是当前日志的数量，一个是允许记录的最大日志的数量。

```
redis> SLOWLOG LEN
(integer) 14
```

清空日志

使用命令 `SLOWLOG RESET` 可以清空 slow log 。

```
redis> SLOWLOG LEN
(integer) 14

redis> SLOWLOG RESET
OK

redis> SLOWLOG LEN
(integer) 0
```

可用版本：

>= 2.2.12

时间复杂度：

O(1)

返回值：

取决于不同命令，返回不同的值。

SYNC

SYNC

用于复制功能(replication)的内部命令。

更多信息请参考 Redis 官网的 Replication 章节 。

可用版本：

>= 1.0.0

时间复杂度：

不明确

返回值：

不明确

```
redis> SYNC
"REDIS0002\xfe\x00\x00\auser_id\xc0\x03\x00\anumbers\xc2\xf3\xe0\x(
(1.90s)
```

TIME

TIME

返回当前服务器时间。

可用版本：

>= 2.6.0

时间复杂度：

O(1)

返回值：

一个包含两个字符串的列表： 第一个字符串是当前时间(以 UNIX 时间戳格式表示)，而第二个字符串是当前这一秒钟已经逝去的微秒数。

```
redis> TIME
1) "1332395997"
2) "952581"
redis> TIME
1) "1332395997"
2) "953148"
```

W3School Memcached 教程

作者：W3School

来源：Memcached 教程

Memcached 入门

Memcached 简介

Memcached是一个自由开源的，高性能，分布式内存对象缓存系统。

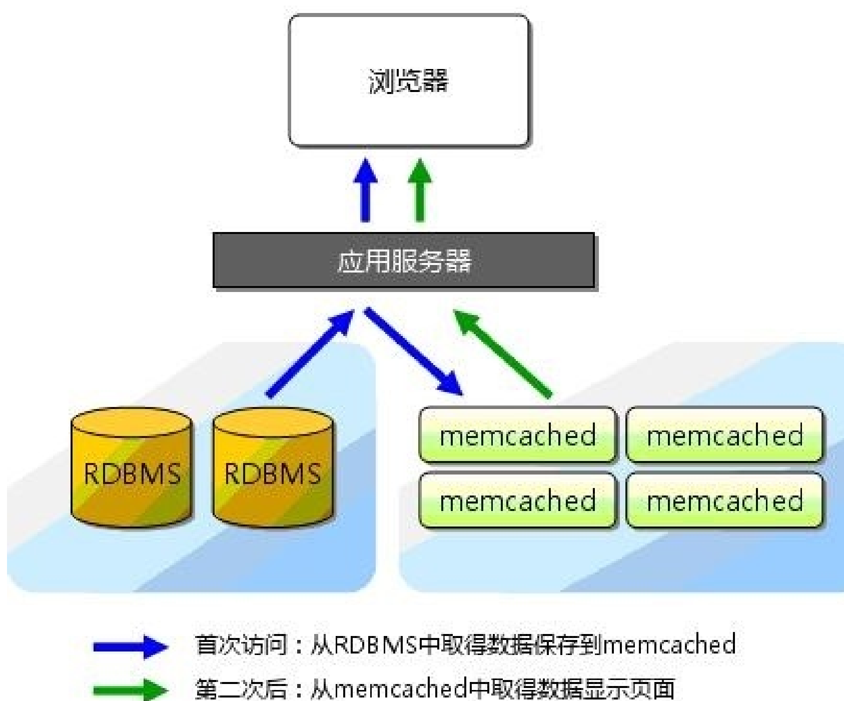
Memcached是以LiveJournal旗下Danga Interactive公司的Brad Fitzpatric为首开发的一款软件。现在已成为mixi、hatena、Facebook、Vox、LiveJournal等众多服务中提高Web应用扩展性的重要因素。

Memcached是一种基于内存的key-value存储，用来存储小块的任意数据（字符串、对象）。这些数据可以是数据库调用、API调用或者是页面渲染的结果。

Memcached简洁而强大。它的简洁设计便于快速开发，减轻开发难度，解决了大数据量缓存的很多问题。它的API兼容大部分流行的开发语言。

本质上，它是一个简洁的key-value存储系统。

一般的使用目的是，通过缓存数据库查询结果，减少数据库访问次数，以提高动态Web应用的速度、提高可扩展性。



Memcached 官网：http://memcached.org/。

特征

memcached作为高速运行的分布式缓存服务器，具有以下的特点。

- 协议简单
- 基于libevent的事件处理
- 内置内存存储方式

- memcached不互相通信的分布式

支持的语言

许多语言都实现了连接memcached的客户端，其中以Perl、PHP为主。仅仅memcached网站上列出的有：

- Perl
- PHP
- Python
- Ruby
- C#
- C/C++
- Lua
- 等等

Memcached 用户

- LiveJournal
- Wikipedia
- Flickr
- Bebo
- Twitter
- Typepad
- Yellowbot
- Youtube
- WordPress.com
- Craigslist
- Mixi

Memcached 安装

Memcached 支持许多平台：Linux、FreeBSD、Solaris、Mac OS，也可以安装在Windows上。

Linux系统安装memcached，首先要先安装libevent库。

```
sudo apt-get install libevent libevent-deve          自动下载安装（Ub
yum install libevent libevent-deve                    自动下载安装
```

安装 Memcached

自动安装

Ubuntu/Debian

```
sudo apt-get install memcached
```

Redhat/Fedora/Centos

```
yum install memcached
```

FreeBSD

```
portmaster databases/memcached
```

源代码安装

从其官方网站（http://memcached.org）下载memcached最新版本。

wget http://memcached.org/latest	下载最新版本
tar -zxvf memcached-1.x.x.tar.gz	解压源码
cd memcached-1.x.x	进入目录
./configure --prefix=/usr/local/memcached	配置
make && make test	编译
sudo make install	安装

Memcached 运行

Memcached命令的运行：

```
$ /usr/local/memcached/bin/memcached -h
```

注意：如果使用自动安装 memcached 命令位于 **/usr/local/bin/memcached**。

启动选项：

- -d是启动一个守护进程；
- -m是分配给Memcache使用的内存数量，单位是MB；
- -u是运行Memcache的用户；
- -l是监听的服务器IP地址，可以有多个地址；
- -p是设置Memcache监听的端口，，最好是1024以上的端口；
- -c是最大运行的并发连接数，默认是1024；
- -P是设置保存Memcache的pid文件。

（1）作为前台程序运行：

从终端输入以下命令，启动memcached:

```
/usr/local/memcached/bin/memcached -p 11211 -m 64m -vv

slab class   1: chunk size     88 perslab 11915

slab class   2: chunk size    112 perslab  9362

slab class   3: chunk size    144 perslab  7281

中间省略

slab class  38: chunk size 391224 perslab     2

slab class  39: chunk size 489032 perslab     2

<23 server listening

<24 send buffer was 110592, now 268435456

<24 server listening (udp)

<24 server listening (udp)

<24 server listening (udp)

<24 server listening (udp)
```

这里显示了调试信息。这样就在前台启动了memcached，监听TCP端口11211，最大内存使用量为64M。调试信息的内容大部分是关于存储的信息。

（2）作为后台服务程序运行：

```
# /usr/local/memcached/bin/memcached -p 11211 -m 64m -d
```

或者

```
/usr/local/memcached/bin/memcached -d -m 64M -u root -l 192.168.0.2
```

Memcached 连接

我们可以通过 telnet 命令并指定主机ip和端口来连接 Memcached 服务。

语法

```
telnet HOST PORT
```

命令中的 **HOST** 和 **PORT** 为运行 Memcached 服务的 IP 和 端口。

实例

以下实例演示了如何连接到 Memcached 服务并执行简单的 set 和 get 命令。

本实例的 Memcached 服务运行的主机为 127.0.0.1（本机）、端口为 11211。

```
telnet 127.0.0.1 11211

Trying 127.0.0.1...

Connected to 127.0.0.1.

Escape character is '^]'.

set foo 0 0 3                                                  保

bar                                                            数

STORED                                                         结

get foo                                                        取

VALUE foo 0 3                                                  数

bar                                                            数

END                                                            结

quit                                                           退
```

Memcached 存储命令

Memcached set 命令

Memcached set 命令用于将 **value(数据值)** 存储在指定的 **key(键)** 中。

如果set的key已经存在，该命令可以更新该key所对应的原来的数据，也就是实现更新的作用。

语法：

set 命令的基本语法格式如下：

```
set key flags exptime bytes [noreply]
value
```

参数说明如下：

- **key**：键值 key-value 结构中的 key，用于查找缓存值。
- **flags**：可以包括键值对的整型参数，客户机使用它存储关于键值对的额外信息。
- **exptime**：在缓存中保存键值对的时间长度（以秒为单位，0 表示永远）
- **bytes**：在缓存中存储的字节数
- **noreply**（可选）： 该参数告知服务器不需要返回数据
- **value**：存储的值（始终位于第二行）（可直接理解为key-value结构中的value）

实例

以下实例中我们设置：

- key → runoob
- flag → 0
- exptime → 900 (以秒为单位)
- bytes → 9 (数据存储的字节数)
- value → memcached

```
set runoob 0 900 9
memcached
STORED

get runoob
VALUE runoob 0 9
memcached

END
```

输出

如果数据设置成功，则输出：

```
STORED
```

输出信息说明：

- **STORED**：保存成功后输出。
- **ERROR**：在保持失败后输出。

Memcached add 命令

Memcached add 命令用于将 **value(数据值)** 存储在指定的 **key(键)** 中。

如果 add 的 key 已经存在，则不会更新数据，之前的值将仍然保持相同，并且您将获得响应 **NOT_STORED**。

语法：

add 命令的基本语法格式如下：

```
add key flags exptime bytes [noreply]
value
```

参数说明如下：

- **key**：键值 key-value 结构中的 key，用于查找缓存值。
- **flags**：可以包括键值对的整型参数，客户机使用它存储关于键值对的额外信息。
- **exptime**：在缓存中保存键值对的时间长度（以秒为单位，0 表示永远）
- **bytes**：在缓存中存储的字节数
- **noreply**（可选）： 该参数告知服务器不需要返回数据
- **value**：存储的值（始终位于第二行）（可直接理解为key-value结构中的value）

实例

以下实例中我们设置：

- key → new_key
- flag → 0
- exptime → 900 (以秒为单位)
- bytes → 10 (数据存储的字节数)
- value → data_value

```
add new_key 0 900 10
data_value
STORED
get new_key
VALUE new_key 0 10
data_value
END
```

输出

如果数据添加成功，则输出：

```
STORED
```

输出信息说明：

- **STORED**：保存成功后输出。
- **NOT_STORED** ：在保持失败后输出。

Memcached replace 命令

Memcached replace 命令用于替换已存在的 **key(键)** 的 **value(数据值)**。

如果 key 不存在，则替换失败，并且您将获得响应 **NOT_STORED**。

语法：

replace 命令的基本语法格式如下：

```
replace key flags exptime bytes [noreply]
value
```

参数说明如下：

- **key**：键值 key-value 结构中的 key，用于查找缓存值。
- **flags**：可以包括键值对的整型参数，客户机使用它存储关于键值对的额外信息。
- **exptime**：在缓存中保存键值对的时间长度（以秒为单位，0 表示永远）
- **bytes**：在缓存中存储的字节数
- **noreply**（可选）： 该参数告知服务器不需要返回数据
- **value**：存储的值（始终位于第二行）（可直接理解为key-value结构中的value）

实例

以下实例中我们设置：

- key → mykey
- flag → 0
- exptime → 900 (以秒为单位)
- bytes → 10 (数据存储的字节数)
- value → data_value

以下实例中我们使用的键位 'mykey' 并存储对应的值 data_value。执行后我们替换相同的 key 的值为 'some_other_value'。

```
add mykey 0 900 10
data_value
STORED
get mykey
VALUE mykey 0 10
data_value
END
replace mykey 0 900 16
some_other_value
get mykey
VALUE mykey 0 16
some_other_value
END
```

输出

如果数据添加成功，则输出：

```
STORED
```

输出信息说明：

- **STORED**：保存成功后输出。
- **NOT_STORED**：执行替换失败后输出。

Memcached append 命令

Memcached append 命令用于向已存在 **key(键)** 的 **value(数据值)** 后面追加数据 。

语法：

append 命令的基本语法格式如下：

```
append key flags exptime bytes [noreply]
value
```

参数说明如下：

- **key**：键值 key-value 结构中的 key，用于查找缓存值。
- **flags**：可以包括键值对的整型参数，客户机使用它存储关于键值对的额外信息 。
- **exptime**：在缓存中保存键值对的时间长度（以秒为单位，0 表示永远）
- **bytes**：在缓存中存储的字节数
- **noreply**（可选）： 该参数告知服务器不需要返回数据
- **value**：存储的值（始终位于第二行）（可直接理解为key-value结构中的value）

实例

实例如下：

- 首先我们在 Memcached 中存储一个键 runoob，其值为 memcached。
- 然后，我们使用 get 命令检索该值。
- 然后，我们使用 **append** 命令在键为 runoob 的值后面追加 "redis"。
- 最后，我们再使用 get 命令检索该值。

```
set runoob 0 900 9
memcached
STORED
get runoob
VALUE runoob 0 14
memcached
END
append runoob 0 900 5
redis
STORED
get runoob
VALUE runoob 0 14
memcachedredis
END
```

输出

如果数据添加成功，则输出：

```
STORED
```

输出信息说明：

- **STORED**：保存成功后输出。
- **NOT_STORED**：该键在 Memcached 上不存在。
- **CLIENT_ERROR**：执行错误。

Memcached prepend 命令

Memcached prepend 命令用于向已存在 **key(键)** 的 **value(数据值)** 前面追加数据。

语法：

prepend 命令的基本语法格式如下：

```
prepend key flags exptime bytes [noreply]
value
```

参数说明如下：

- **key**：键值 key-value 结构中的 key，用于查找缓存值。
- **flags**：可以包括键值对的整型参数，客户机使用它存储关于键值对的额外信息。
- **exptime**：在缓存中保存键值对的时间长度（以秒为单位，0 表示永远）
- **bytes**：在缓存中存储的字节数
- **noreply**（可选）： 该参数告知服务器不需要返回数据
- **value**：存储的值（始终位于第二行）（可直接理解为key-value结构中的value）

实例

实例如下：

- 首先我们在 Memcached 中存储一个键 runoob，其值为 memcached。
- 然后，我们使用 get 命令检索该值。
- 然后，我们使用 **prepend** 命令在键为 runoob 的值后面追加 "redis"。
- 最后，我们再使用 get 命令检索该值。

```
set runoob 0 900 9
memcached
STORED
get runoob
VALUE runoob 0 14
memcached
END
prepend runoob 0 900 5
redis
STORED
get runoob
VALUE runoob 0 14
redismemcached
END
```

输出

如果数据添加成功，则输出：

```
STORED
```

输出信息说明：

- **STORED**：保存成功后输出。
- **NOT_STORED**：该键在 Memcached 上不存在。
- **CLIENT_ERROR**：执行错误。

Memcached CAS 命令

Memcached CAS（Check-And-Set 或 Compare-And-Swap） 命令用于执行一个"检查并设置"的操作

它仅在当前客户端最后一次取值后，该key 对应的值没有被其他客户端修改的情况下， 才能够将值写入。

检查是通过cas_token参数进行的， 这个参数是Memcach指定给已经存在的元素的一个唯一的64位值。

语法：

CAS 命令的基本语法格式如下：

```
cas key flags exptime bytes unique_cas_token [noreply]
value
```

参数说明如下：

- **key**：键值 key-value 结构中的 key，用于查找缓存值。
- **flags**：可以包括键值对的整型参数，客户机使用它存储关于键值对的额外信息。
- **exptime**：在缓存中保存键值对的时间长度（以秒为单位，0 表示永远）
- **bytes**：在缓存中存储的字节数
- **unique_cas_token**通过 gets 命令获取的一个唯一的64位值。
- **noreply**（可选）： 该参数告知服务器不需要返回数据
- **value**：存储的值（始终位于第二行）（可直接理解为key-value结构中的value）

实例

要在 Memcached 上使用 CAS 命令，你需要从 Memcached 服务商通过 gets 命令获取令牌（token）。

gets 命令的功能类似于基本的 get 命令。两个命令之间的差异在于，gets 返回的信息稍微多一些：64 位的整型值非常像名称/值对的 "版本" 标识符。

实例步骤如下：

- 如果没有设置唯一令牌，则 CAS 命令执行错误。
- 如果键 key 不存在，执行失败。
- 添加键值对。
- 通过 gets 命令获取唯一令牌。
- 使用 cas 命令更新数据

- 使用 get 命令查看数据是否更新

```
cas tp 0 900 9
ERROR             <? 缺少 token

cas tp 0 900 9 2
memcached
NOT_FOUND         <? 键 tp 不存在

set tp 0 900 9
memcached
STORED

gets tp
VALUE tp 0 9 1
memcached
END

cas tp 0 900 5 1
redis
STORED

get tp
VALUE tp 0 5
redis
END
```

输出

如果数据添加成功，则输出：

```
STORED
```

输出信息说明：

- **STORED**：保存成功后输出。
- **ERROR**：保存出错或语法错误。
- **EXISTS**：在最后一次取值后另外一个用户也在更新该数据。
- **NOT_FOUND**：Memcached 服务上不存在该键值。

Memcached 查找命令

Memcached get 命令

Memcached get 命令获取存储在 **key(键)** 中的 **value(数据值)**， 如果 key 不存在，则返回空。

语法：

get 命令的基本语法格式如下：

```
get key
```

多个 key 使用空格隔开，如下:

```
get key1 key2 key3
```

参数说明如下：

- **key**：键值 key-value 结构中的 key，用于查找缓存值。

实例

在以下实例中，我们使用 runoob 作为 key，过期时间设置为 900 秒。

```
set runoob 0 900 9
memcached
STORED
get runoob
VALUE runoob 0 9
memcached
END
```

Memcached gets 命令

Memcached gets 命令获取带有 CAS 令牌存 的 **value(数据值)** ，如果 key 不存在，则返回空。

语法：

gets 命令的基本语法格式如下：

```
gets key
```

多个 key 使用空格隔开，如下:

```
gets key1 key2 key3
```

参数说明如下：

- **key**：键值 key-value 结构中的 key，用于查找缓存值。

实例

在以下实例中，我们使用 runoob 作为 key，过期时间设置为 900 秒。

```
set runoob 0 900 9
memcached
STORED
gets runoob
VALUE runoob 0 9 1
memcached
END
```

在 使用 gets 命令的输出结果中，在最后一列的数字 1 代表了 key 为 runoob 的 CAS 令牌。

Memcached delete 命令

Memcached delete 命令用于删除已存在的 key(键)。

语法：

delete 命令的基本语法格式如下：

```
delete key [noreply]
```

多个 key 使用空格隔开，如下:

```
gets key1 key2 key3
```

参数说明如下：

- **key**：键值 key-value 结构中的 key，用于查找缓存值。
- **noreply**（可选）： 该参数告知服务器不需要返回数据

实例

在以下实例中，我们使用 runoob 作为 key，过期时间设置为 900 秒。之后我们使用 delete 命令删除该 key。

```
set runoob 0 900 9
memcached
STORED
get runoob
VALUE runoob 0 9
memcached
END
delete runoob
DELETED
get runoob
END
delete runoob
NOT_FOUND
```

输出

输出信息说明：

- **DELETED**：删除成功。

- **ERROR**：语法错误或删除失败。
- **NOT_FOUND**：key 不存在。

Memcached incr 与 decr 命令

Memcached incr 与 decr 命令用于对已存在的 key(键) 的数字值进行自增或自减操作。

incr 与 decr 命令操作的数据必须是十进制的32位无符号整数。

如果 key 不存在返回 **NOT_FOUND**，如果键的值不为数字，则返回 **CLIENT_ERROR**，其他错误返回 **ERROR**。

incr 命令

语法：

incr 命令的基本语法格式如下：

```
incr key increment_value
```

参数说明如下：

- **key**：键值 key-value 结构中的 key，用于查找缓存值。
- **increment_value**： 增加的数值。

实例

在以下实例中，我们使用 visitors 作为 key，初始值为 10，之后进行加 5 操作。

```
set visitors 0 900 2
10
STORED
get visitors
VALUE visitors 0 2
10
END
incr visitors 5
15
get visitors
VALUE visitors 0 2
15
END
```

输出

输出信息说明：

- **NOT_FOUND**：key 不存在。
- **CLIENT_ERROR**：自增值不是对象。
- **ERROR**其他错误，如语法错误等。

decr 命令

decr 命令的基本语法格式如下：

```
decr key decrement_value
```

参数说明如下：

- **key**：键值 key-value 结构中的 key，用于查找缓存值。
- **decrement_value**： 减少的数值。

实例

```
set visitors 0 900 2
10
STORED
get visitors
VALUE visitors 0 2
10
END
decr visitors 5
5
get visitors
VALUE visitors 0 1
5
END
```

在以下实例中，我们使用 visitors 作为 key，初始值为 10，之后进行减 5 操作。

输出

输出信息说明：

- **NOT_FOUND**：key 不存在。
- **CLIENT_ERROR**：自增值不是对象。
- **ERROR**其他错误，如语法错误等。

Memcached 统计命令

Memcached stats 命令

Memcached stats 命令用于返回统计信息例如 PID(进程号)、版本号、连接数等。

语法：

stats 命令的基本语法格式如下：

```
stats
```

实例

在以下实例中，我们使用了 stats 命令来输出 Memcached 服务信息。

```
stats
STAT pid 1162
STAT uptime 5022
STAT time 1415208270
STAT version 1.4.14
STAT libevent 2.0.19-stable
STAT pointer_size 64
STAT rusage_user 0.096006
STAT rusage_system 0.152009
STAT curr_connections 5
STAT total_connections 6
STAT connection_structures 6
STAT reserved_fds 20
STAT cmd_get 6
STAT cmd_set 4
STAT cmd_flush 0
STAT cmd_touch 0
STAT get_hits 4
STAT get_misses 2
STAT delete_misses 1
STAT delete_hits 1
STAT incr_misses 2
STAT incr_hits 1
STAT decr_misses 0
STAT decr_hits 1
STAT cas_misses 0
STAT cas_hits 0
STAT cas_badval 0
STAT touch_hits 0
STAT touch_misses 0
STAT auth_cmds 0
STAT auth_errors 0
STAT bytes_read 262
STAT bytes_written 313
STAT limit_maxbytes 67108864
STAT accepting_conns 1
STAT listen_disabled_num 0
STAT threads 4
STAT conn_yields 0
STAT hash_power_level 16
STAT hash_bytes 524288
STAT hash_is_expanding 0
STAT expired_unfetched 1
STAT evicted_unfetched 0
STAT bytes 142
STAT curr_items 2
STAT total_items 6
STAT evictions 0
STAT reclaimed 1
END
```

这里显示了很多状态信息，下边详细解释每个状态项：

- **pid**： memcache服务器进程ID
- **uptime**：服务器已运行秒数
- **time**：服务器当前Unix时间戳
- **version**：memcache版本
- **pointer_size**：操作系统指针大小
- **rusage_user**：进程累计用户时间
- **rusage_system**：进程累计系统时间
- **curr_connections**：当前连接数量
- **total_connections**：Memcached运行以来连接总数
- **connection_structures**：Memcached分配的连接结构数量
- **cmd_get**：get命令请求次数
- **cmd_set**：set命令请求次数
- **cmd_flush**：flush命令请求次数
- **get_hits**：get命令命中次数
- **get_misses**：get命令未命中次数
- **delete_misses**：delete命令未命中次数
- **delete_hits**：delete命令命中次数
- **incr_misses**：incr命令未命中次数
- **incr_hits**：incr命令命中次数
- **decr_misses**：decr命令未命中次数
- **decr_hits**：decr命令命中次数
- **cas_misses**：cas命令未命中次数
- **cas_hits**：cas命令命中次数
- **cas_badval**：使用擦拭次数
- **auth_cmds**：认证命令处理的次数
- **auth_errors**：认证失败数目
- **bytes_read**：读取总字节数
- **bytes_written**：发送总字节数
- **limit_maxbytes**：分配的内存总大小（字节）
- **accepting_conns**：服务器是否达到过最大连接（0/1）
- **listen_disabled_num**：失效的监听数
- **threads**：当前线程数
- **conn_yields**：连接操作主动放弃数目
- **bytes**：当前存储占用的字节数
- **curr_items**：当前存储的数据总数
- **total_items**：启动以来存储的数据总数
- **evictions**：LRU释放的对象数目
- **reclaimed**：已过期的数据条目来存储新数据的数目

Memcached stats items 命令

Memcached stats items 命令用于显示各个 slab 中 item 的数目和存储时长(最后一次访问距离现在的秒数)。

语法：

stats items 命令的基本语法格式如下：

```
stats items
```

实例

```
stats items
STAT items:1:number 1
STAT items:1:age 7
STAT items:1:evicted 0
STAT items:1:evicted_nonzero 0
STAT items:1:evicted_time 0
STAT items:1:outofmemory 0
STAT items:1:tailrepairs 0
STAT items:1:reclaimed 0
STAT items:1:expired_unfetched 0
STAT items:1:evicted_unfetched 0
END
```

Memcached stats slabs 命令

Memcached stats slabs 命令用于显示各个slab的信息，包括chunk的大小、数目、使用情况等。

语法：

stats slabs 命令的基本语法格式如下：

```
stats slabs
```

实例

```
stats slabs
STAT 1:chunk_size 96
STAT 1:chunks_per_page 10922
STAT 1:total_pages 1
STAT 1:total_chunks 10922
STAT 1:used_chunks 1
STAT 1:free_chunks 10921
STAT 1:free_chunks_end 0
STAT 1:mem_requested 71
STAT 1:get_hits 0
STAT 1:cmd_set 1
STAT 1:delete_hits 0
STAT 1:incr_hits 0
STAT 1:decr_hits 0
STAT 1:cas_hits 0
STAT 1:cas_badval 0
STAT 1:touch_hits 0
STAT active_slabs 1
STAT total_malloced 1048512
END
```

Memcached stats sizes 命令

Memcached stats sizes 命令用于显示所有item的大小和个数。

该信息返回两列，第一列是 item 的大小，第二列是 item 的个数。

语法：

stats sizes 命令的基本语法格式如下：

```
stats sizes
```

实例

```
stats sizes
STAT 96 1
END
```

Memcached flush_all 命令

Memcached flush_all 命令用于用于清理缓存中的所有 **key=>value(键=>值)** 对。

该命令提供了一个可选参数 **time**，用于在制定的时间后执行清理缓存操作。

语法：

flush_all 命令的基本语法格式如下：

```
flush_all [time] [noreply]
```

实例

清理缓存:

```
set runoob 0 900 9
memcached
STORED
get runoob
VALUE runoob 0 9
memcached
END
flush_all
OK
get runoob
END
```

Memcached 实例

Java 连接 Memcached 服务

使用 Java 程序连接 Memcached，需要在你的 classpath 中添加 Memcached jar 包。

本站 jar 包下载地址：spymemcached-2.10.3.jar。Google Code jar 包下载地址：spymemcached-2.10.3.jar（需要翻墙）。

以下程序假定 Memcached 服务的主机为 127.0.0.1，端口为 11211。

连接实例

Java 连接 Memcached

```
import net.spy.memcached.MemcachedClient;
import java.net.*;

public class MemcachedJava {
   public static void main(String[] args) {
      try{
         // 本地连接 Memcached 服务
         MemcachedClient mcc = new MemcachedClient(new InetSocketAd
         System.out.println("Connection to server sucessful.");

         // 关闭连接
         mcc.shutdown();

      }catch(Exception ex){
         System.out.println( ex.getMessage() );
      }
   }
}
```

该程序中我们使用 InetSocketAddress 连接 IP 为 127.0.0.1 端口 为 11211 的 memcached 服务。

执行以上代码，如果连接成功会输出以下信息：

```
Connection to server successful.
```

set 操作实例

以下使用 java.util.concurrent.Future 来存储数据

```
import java.net.InetSocketAddress;
import java.util.concurrent.Future;

import net.spy.memcached.MemcachedClient;

public class MemcachedJava {
   public static void main(String[] args) {

      try{
         // 连接本地的 Memcached 服务
         MemcachedClient mcc = new MemcachedClient(new InetSocketAd
         System.out.println("Connection to server sucessful.");

         // 存储数据
         Future fo = mcc.set("runoob", 900, "Free Education");

         // 查看存储状态
         System.out.println("set status:" + fo.get());

         // 输出值
         System.out.println("runoob value in cache - " + mcc.get("r

         // 关闭连接
         mcc.shutdown();

      }catch(Exception ex){
         System.out.println( ex.getMessage() );
      }
   }
}
```

执行程序，输出结果为：

```
Connection to server successful.
set status:true
runoob value in cache - Free Education
```

add 操作实例

```
import java.net.InetSocketAddress;
import java.util.concurrent.Future;

import net.spy.memcached.MemcachedClient;

public class MemcachedJava {
   public static void main(String[] args) {

      try{

         // 连接本地的 Memcached 服务
         MemcachedClient mcc = new MemcachedClient(new InetSocketAd
         System.out.println("Connection to server sucessful.");

         // 添加数据
         Future fo = mcc.set("runoob", 900, "Free Education");

         // 打印状态
         System.out.println("set status:" + fo.get());

         // 输出
         System.out.println("runoob value in cache - " + mcc.get("r

         // 添加
         Future fo = mcc.add("runoob", 900, "memcached");

         // 打印状态
         System.out.println("add status:" + fo.get());

         // 添加新key
         fo = mcc.add("codingground", 900, "All Free Compilers");

         // 打印状态
         System.out.println("add status:" + fo.get());

         // 输出
         System.out.println("codingground value in cache - " + mcc.

         // 关闭连接
         mcc.shutdown();

      }catch(Exception ex){
         System.out.println(ex.getMessage());
      }
   }
}
```

replace 操作实例

```
import java.net.InetSocketAddress;
import java.util.concurrent.Future;

import net.spy.memcached.MemcachedClient;

public class MemcachedJava {
   public static void main(String[] args) {

      try {
         //连接本地的 Memcached 服务
         MemcachedClient mcc = new MemcachedClient(new InetSocketAd
         System.out.println("Connection to server sucessful.");

         // 添加第一个 key=》value 对
         Future fo = mcc.set("runoob", 900, "Free Education");

         // 输出执行 add 方法后的状态
         System.out.println("add status:" + fo.get());

         // 获取键对应的值
         System.out.println("runoob value in cache - " + mcc.get("r

         // 添加新的 key
         fo = mcc.replace("runoob", 900, "Largest Tutorials' Librar

         // 输出执行 set 方法后的状态
         System.out.println("replace status:" + fo.get());

         // 获取键对应的值
         System.out.println("runoob value in cache - " + mcc.get("r

         // 关闭连接
         mcc.shutdown();

      }catch(Exception ex){
         System.out.println( ex.getMessage() );
      }
   }
}
```

append 操作实例

```
import java.net.InetSocketAddress;
import java.util.concurrent.Future;

import net.spy.memcached.MemcachedClient;

public class MemcachedJava {
   public static void main(String[] args) {

      try{

         // 连接本地的 Memcached 服务
         MemcachedClient mcc = new MemcachedClient(new InetSocketAd
         System.out.println("Connection to server sucessful.");

         // 添加数据
         Future fo = mcc.set("runoob", 900, "Free Education");

         // 输出执行 set 方法后的状态
         System.out.println("set status:" + fo.get());

         // 获取键对应的值
         System.out.println("runoob value in cache - " + mcc.get("r

         // 对存在的key进行数据添加操作
         Future fo = mcc.append("runoob", 900, " for All");

         // 输出执行 set 方法后的状态
         System.out.println("append status:" + fo.get());

         // 获取键对应的值
         System.out.println("runoob value in cache - " + mcc.get("c

         // 关闭连接
         mcc.shutdown();

      }catch(Exception ex)
         System.out.println(ex.getMessage());
   }
}
```

prepend 操作实例

```
import java.net.InetSocketAddress;
import java.util.concurrent.Future;

import net.spy.memcached.MemcachedClient;

public class MemcachedJava {
   public static void main(String[] args) {

      try{

         // 连接本地的 Memcached 服务
         MemcachedClient mcc = new MemcachedClient(new InetSocketAc
         System.out.println("Connection to server sucessful.");

         // 添加数据
         Future fo = mcc.set("runoob", 900, "Education for All");

         // 输出执行 set 方法后的状态
         System.out.println("set status:" + fo.get());

         // 获取键对应的值
         System.out.println("runoob value in cache - " + mcc.get("r

         // 对存在的key进行数据添加操作
         Future fo = mcc.prepend("runoob", 900, "Free ");

         // 输出执行 set 方法后的状态
         System.out.println("prepend status:" + fo.get());

         // 获取键对应的值
         System.out.println("runoob value in cache - " + mcc.get("c

         // 关闭连接
         mcc.shutdown();

      }catch(Exception ex)
         System.out.println(ex.getMessage());
   }
}
```

CAS 操作实例

```
import java.net.InetSocketAddress;
import java.util.concurrent.Future;

import net.spy.memcached.CASValue;
import net.spy.memcached.CASResponse;
import net.spy.memcached.MemcachedClient;

public class MemcachedJava {
   public static void main(String[] args) {

      try{

         // 连接本地的 Memcached 服务
         MemcachedClient mcc = new MemcachedClient(new InetSocketAd
         System.out.println("Connection to server sucessful.");

         // 添加数据
         Future fo = mcc.set("runoob", 900, "Free Education");

         // 输出执行 set 方法后的状态
         System.out.println("set status:" + fo.get());

         // 使用 get 方法获取数据
         System.out.println("runoob value in cache - " + mcc.get("r

         // 通过 gets 方法获取 CAS token（令牌）
         CASValue casValue = mcc.gets("runoob");

         // 输出 CAS token（令牌） 值
         System.out.println("CAS token - " + casValue);

         // 尝试使用cas方法来更新数据
         CASResponse casresp = mcc.cas("runoob", casValue.getCas(),

         // 输出 CAS 响应信息
         System.out.println("CAS Response - " + casresp);

         // 输出值
         System.out.println("runoob value in cache - " + mcc.get("r

         // 关闭连接
         mcc.shutdown();

      }catch(Exception ex)
         System.out.println(ex.getMessage());
   }
}
```

get 操作实例

```
import java.net.InetSocketAddress;
import java.util.concurrent.Future;

import net.spy.memcached.MemcachedClient;

public class MemcachedJava {
   public static void main(String[] args) {

      try{

         // 连接本地的 Memcached 服务
         MemcachedClient mcc = new MemcachedClient(new InetSocketAd
         System.out.println("Connection to server sucessful.");

         // 添加数据
         Future fo = mcc.set("runoob", 900, "Free Education");

         // 输出执行 set 方法后的状态
         System.out.println("set status:" + fo.get());

         // 使用 get 方法获取数据
         System.out.println("runoob value in cache - " + mcc.get("r

         // 关闭连接
         mcc.shutdown();

      }catch(Exception ex)
         System.out.println(ex.getMessage());
   }
}
```

gets 操作实例、CAS

```
import java.net.InetSocketAddress;
import java.util.concurrent.Future;

import net.spy.memcached.CASValue;
import net.spy.memcached.CASResponse;
import net.spy.memcached.MemcachedClient;

public class MemcachedJava {
   public static void main(String[] args) {

      try{

         // 连接本地的 Memcached 服务
         MemcachedClient mcc = new MemcachedClient(new InetSocketAd
         System.out.println("Connection to server sucessful.");

         // 添加数据
         Future fo = mcc.set("runoob", 900, "Free Education");

         // 输出执行 set 方法后的状态
         System.out.println("set status:" + fo.get());

         // 从缓存中获取键为 runoob 的值
         System.out.println("runoob value in cache - " + mcc.get("r

         // 通过 gets 方法获取 CAS token（令牌）
         CASValue casValue = mcc.gets("runoob");

         // 输出 CAS token（令牌） 值
         System.out.println("CAS value in cache - " + casValue);

         // 关闭连接
         mcc.shutdown();

      }catch(Exception ex)
         System.out.println(ex.getMessage());
   }
}
```

delete 操作实例

```
import java.net.InetSocketAddress;
import java.util.concurrent.Future;

import net.spy.memcached.MemcachedClient;

public class MemcachedJava {
   public static void main(String[] args) {

      try{

         // 连接本地的 Memcached 服务
         MemcachedClient mcc = new MemcachedClient(new InetSocketAd
         System.out.println("Connection to server sucessful.");

         // 添加数据
         Future fo = mcc.set("runoob", 900, "World's largest online

         // 输出执行 set 方法后的状态
         System.out.println("set status:" + fo.get());

         // 获取键对应的值
         System.out.println("runoob value in cache - " + mcc.get("r

         // 对存在的key进行数据添加操作
         Future fo = mcc.delete("runoob");

         // 输出执行 delete 方法后的状态
         System.out.println("delete status:" + fo.get());

         // 获取键对应的值
         System.out.println("runoob value in cache - " + mcc.get("c

         // 关闭连接
         mcc.shutdown();

      }catch(Exception ex)
         System.out.println(ex.getMessage());
   }
}
```

Incr/Decr 操作实例

```
import java.net.InetSocketAddress;
import java.util.concurrent.Future;

import net.spy.memcached.MemcachedClient;

public class MemcachedJava {
   public static void main(String[] args) {

      try{

         // 连接本地的 Memcached 服务
         MemcachedClient mcc = new MemcachedClient(new InetSocketAd
         System.out.println("Connection to server sucessful.");

         // 添加数字值
         Future fo = mcc.set("number", 900, "1000");

         // 输出执行 set 方法后的状态
         System.out.println("set status:" + fo.get());

         // 获取键对应的值
         System.out.println("value in cache - " + mcc.get("number")

         // 自增并输出
         System.out.println("value in cache after increment - " + m

         // 自减并输出
         System.out.println("value in cache after decrement - " + m

         // 关闭连接
         mcc.shutdown();

      }catch(Exception ex)
         System.out.println(ex.getMessage());
   }
}
```

PHP 连接 Memcached 服务

在前面章节中我们已经介绍了如何安装 Memcached 服务，接下来我们为大家介绍 PHP 如何使用 Memcached 服务。

PHP Memcache 扩展安装

PHP Memcache 扩展包下载地址：http://pecl.php.net/package/memcache，你可以下载最新稳定包(stable)。

```
wget http://pecl.php.net/get/memcache-2.2.7.tgz
tar -zxvf memcache-2.2.7.tgz
cd memcache-2.2.7
/usr/local/php/bin/phpize
./configure --with-php-config=/usr/local/php/bin/php-config
make && make install
```

注意：/usr/local/php/ 为php的安装路径，需要根据你安装的实际目录调整。

安装成功后会显示你的memcache.so扩展的位置，比如我的：

```
Installing shared extensions:     /usr/local/php/lib/php/extensions
```

最后我们需要把这个扩展添加到php中，打开你的php.ini文件在最后添加以下内容：

```
[Memcache]
extension_dir = "/usr/local/php/lib/php/extensions/no-debug-non-zts
extension = memcache.so
```

添加完后 重新启动php,我使用的是nginx+php-fpm进程所以命令如下：

```
kill -USR2 `cat /usr/local/php/var/run/php-fpm.pid`
```

如果是apache的使用以下命令:

```
/usr/local/apache2/bin/apachectl restart
```

检查安装结果

```
/usr/local/php/bin/php -m | grep memcache
```

安装成功会输出：memcache。

或者通过浏览器访问 phpinfo() 函数来查看，如下图：

memcache

memcache support	enabled
Active persistent connections	0
Version	2.2.7
Revision	$Revision: 327750 $

Directive	Local Value	Master Value
memcache.allow_failover	1	1
memcache.chunk_size	8192	8192
memcache.default_port	11211	11211
memcache.default_timeout_ms	1000	1000
memcache.hash_function	crc32	crc32
memcache.hash_strategy	standard	standard
memcache.max_failover_attempts	20	20

PHP 连接 Memcached

```
<?php
$memcache = new Memcache;             //创建一个memcache对象
$memcache->connect('localhost', 11211) or die ("Could not connect")
$memcache->set('key', 'test');        //设置一个变量到内存中，名称是key
$get_value = $memcache->get('key');   //从内存中取出key的值
echo $get_value;
?>
```

更多 PHP 操作 Memcached 请参阅：
http://php.net/manual/zh/book.memcache.php

免责声明

版权信息

菜鸟教程（www.runoob.com）刊载的所有内容，包括文字、图片、音频、视频、软件、程序、以及网页版式设计等均在网上搜集。

菜鸟教程提供的内容仅用于个人学习、研究或欣赏。我们不保证内容的正确性。通过使用本站内容随之而来的风险与本站无关

访问者可将本网站提供的内容或服务用于个人学习、研究或欣赏，以及其他非商业性或非盈利性用途，但同时应遵守著作权法及其他相关法律的规定，不得侵犯本网站及相关权利人的合法权利。

本网站内容原作者如不愿意在本网站刊登内容，请及时通知本站，予以删除。

本网站的编程技术内容大部分翻译自 W3Schools Online Web Tutorials 与 Tutorialspoint, 内容原作者如不愿意在本网站刊登内容，请及时通知本站，予以删除。

链接到菜鸟教程

任何网站都可以链接到菜鸟教程的任何页面。

如果您需要在对少量内容进行引用，请务必在引用该内容的页面添加指向被引用页面的链接。

保证

菜鸟教程不提供任何形式的保证。所有与使用本站相关的直接风险均由用户承担。菜鸟教程提供的所有代码均为实例，并不对性能、适用性、适销性或/及其他方面提供任何保证。

菜鸟教程的内容可能包含不准确性或错误。菜鸟教程不对本网站及其内容的准确性进行保证。如果您发现本站点及其内容包含错误，请联系我们以便这些错误得到及时的更正：429240967@qq.com

您的行为

当您使用本站点时，说明您已经同意并接受本页面的所有信息。

联系方式

联系邮箱：429240967@qq.com（相关事务请发函至该邮箱）